FINDING AND REMEDYING HIGH-LEVEL SECURITY ISSUES IN BINARY CODE

A Thesis

Presented to

The Academic Faculty

By

David B. Dewey

In Partial Fulfillment

Of the Requirements for the Degree

Doctor of Philosophy in Computer Science

Georgia Institute of Technology

December 2015

FINDING AND REMEDYING HIGH-LEVEL SECURITY ISSUES IN BINARY CODE

For my wife, Katie,

and my daughters, Emma and Sarah.

While it will be my name on the degree,

we all earned this together.

ACKNOWLEDGEMENTS

Of all of the people I have met throughout my professional career, there is one that I admire the most. Terry Nelms is by far the most intelligent, yet humble engineer I have ever met. In the spring of 2009, Terry told me he was going to pursue a PhD at Georgia Tech. Then he said something I never expected, “You should do it, too.” I had never considered going back to school; specially not for something as intimidating as a PhD. But there was Terry Nelms telling me he thought I could do it. How could I *not* do it?

Terry introduced me to Mustaque Ahamad. Since that time, Mustaque has been extremely influential in my life inside and outside of Georgia Tech. He helped me navigate the enrollment process, introduced me to many great people at IBM research, added me to the GTISC advisory board, and now I work with Mustaque at Pindrop Security. In every encounter with Mustaque, I find some new incredibly impressive facet of his background and talents.

Mustaque suggested Jonathon Giffin as my advisor. Jon was a great fit. We shared so much in terms of what motivated and interested us. Jon really forced me to focus on the science of code analysis. He completely changed my outlook on how programs and programming languages should be analyzed. When he chose to move into industry, I couldn’t have given a higher recommendation to my colleagues at Fortify Software. Jon was, without a doubt, a perfect fit for that team.

My first class at Georgia Tech was Patrick Traynor’s network security class. As this was my first graduate-level course, and I was coming to it with ten years of information security experience, I was a little skeptical. I was immediately impressed with Patrick, and Georgia Tech as a whole. The way Patrick conducted his class, the expectations of the students, and the depth of the material made it clear to me why Georgia Tech was rated so highly. The most impressive thing I saw Patrick do was when he caught a student cheating. He forced the student to withdraw failing from the course. Coming from the Virginia Military Institute, honor is extremely important to me. We often think that true honor does not exist outside of VMI, but Patrick proved that it does.

When I needed to find a new advisor, there was no question who I was going ask. Patrick has been an incredible mentor. He forced me to focus on, and improve, my weaknesses. Before working with Patrick, I often struggled to take constructive criticism. Rather than treating it like an emotional deficiency, Patrick helped me logic my way through why constructive criticism is so important. He also forced me to become a better writer. I'm still not where I would like to be (more on that later), but I'm definitely better than I was.

I want to thank my committee. I know my research topic is well outside the common areas of focus in GTISC. Everyone has been so accommodating in learning more about the area so they can provide relevant feedback. Special thanks to Patrick Traynor, Mustaque Ahamad, Jonathon Giffin, Bill Harris, and Sasha Boldyreva.

My research would never be where it is without Mark Dowd and Ryan Smith. Mark and Ryan have the ability to dig into the deepest levels of technology and learn everything there is to know about a topic without any documentation or guiding road map of any kind. In short, they are the best at what they do. I was lucky to get to work with them.

Finally, I would be remiss in not thanking one of my co-authors: Brad Reaves. Brad is, by far, the best writer I have ever known. I learned so much from him by working on papers together. There were so many times that I would write something, think it was great, and he would make it so much better. By looking at the ways he changed what I had written I learned a number of skills that have improved my own writing going forward.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

C++ and Microsoft's Component Object Model (COM) are examples of a high-level language and development framework that were built on top of the lower-level, primitive language, C. C was never designed to support concepts like object orientation, type enforcement, and language independence. Further, these languages and frameworks are designed to compile and run directly on the processor where these concepts are also not supported. Other high-level languages that do support these concepts make use of a runtime or virtual machine to create a computing model to suit their needs. By forcing these high-level concepts into a primitive computing model, many security issues have been introduced. Existing binary-level security analysis tools and runtime enforcement frameworks operate at the lowest level of context. As such, they struggle to detect and remedy higher-level security issues. In this dissertation, a framework for elevating the context of binary code is presented. By bringing the context for analysis closer to where these security issues are introduced, this framework allows for higher-level analyses and enforcement frameworks to be developed.

Chapter I

INTRODUCTION

As is often the case, when a fundamental technology is adapted and abstracted for higher-level concepts, security issues arise. Take for example, the use of HTTP as the transport protocol for web applications. HTTP — a stateless protocol — has to be pretty drastically misused to support stateful web applications, leading to myriad security issues. This same concept exists at the programming language level. C served as the primary programming language for many years. C is an extremely low-level language, providing very little abstraction from processor-level primitives. With that, there is a very close coupling between the computing model of the processor and the programming model of C. C types are one-to-one representations of types available at the processor level. Memory allocations and accesses represent direct access to the memory used to store data (of course, ignoring virtual memory layout). Because of this close coupling to the processor, C provides developers with extremely powerful tools that allow for the extreme optimization of their code. Of course, with great power comes great responsibility. C developers shoulder the responsibility of enforcing types, ensuring proper memory accesses, and general error checking.

As applications became more and more complex, and programming became a necessary skill for the masses, C became an unwieldy tool that did not meet the needs for many developers. The advent of object oriented programming, type-enforcement, and language independence required higher-level language constructs than those provided in C. The first steps toward providing these constructs was to build higher-level languages on top of C. In fact, early C++ compilers actually translated the C++ code into C. C++, for example, builds on top of the C `struct` type. The `struct` type was not intended to support object orientation, but by adding some compile-time modifications, it became the foundation for objects in C++. Extending this concept to yet another level of abstraction, Microsoft's Component Object Model (COM) is built on top of C++. COM provides a framework for self-describing, language independent, type-enforced object orientation with a standard interface. Note that none of the properties of COM just listed are actually properties of, or enforceable in, C++.

Much like the HTTP example, using C as the fundamental building block for C++ and COM has led to security issues. Because there is no type enforcement at the C level (because it is not enforced by the processor), it is impossible to build proper type enforcement for C++ of COM. Because there is no implicit reference counting for memory allocations, issues like memory leaks and use-after-free conditions are possible. As will be seen throughout this dissertation, oftentimes, the inventors of these higher-level languages try to fix these exact kinds of issues. However, this is only so much that can be done since the primary building block was not designed to support these concepts. In fact, as we will show, it is often the attempt to fix these issues that leads to other security issues.

In this work, we discuss the details of how these adaptations and abstractions have led to security issues. We cover issues that are introduced into these higher-level constructs at the time objects are instantiated, as they are used during their lifetime, and as they are being deleted. In the case of C++, security issues arise at instantiation time when constructors fail to properly execute. During their lifetime, we see issues with type confusion and the security issues associated with treating an object of the wrong type. Because object deletion is the onus of the developer, we find problems that arise causing use-after-free conditions. In the case of COM, we show the security issues that arise when instantiation fails. Throughout the lifetime of a COM object, we will demonstrate issues related to type confusion, the copying of objects, and a trust transitivity issue that exists in the COM security infrastructure. As COM objects are deleted, we will show issues that occur when deletion is handled incorrectly.

Software security vulnerabilities are as old as software itself. Because of this, many security analysis frameworks and runtime enforcement engines have been developed to try to eliminate and/or mitigate these problems. Static analysis frameworks that function on source code have shown significant progress toward reducing the number of vulnerabilities that are released into production code. However, many applications are still commonplace on modern-day operating systems that were developed prior to the widespread adoption of such source code analysis tools. Additionally, not every software vendor employs such tools, or maintains strict secure coding practices throughout their software development life cycle. With that, when closed source applications are deployed in the world, the user has no other way to evaluate their true security risk than analyzing the binary code. Further, if security issues are discovered, runtime enforcement engines (e.g., host-based IDS, anti-virus systems, etc.) must interface with these applications through binary insertion.

The static analysis frameworks that were developed for analyzing source code have a major advantage over those that much interface with the resultant binary code: *context*. As high-level code is compiled, much of the context present in the source code is lost. In essence, the high-level constructs present in C++ and COM have to be translated into some low-level contextual equivalent that can be represented and operated on by the processor. As discussed, the processor has no concept of an object (C++ or COM object). The process of compilation translates these higher-level constructs into their lower-level counterparts. Because the goal of this work is to address security issues introduced into applications by high-level language constructs, operating at the low-level context provided by binaries is insufficient.

1.1 Thesis Statement

The work presented in this dissertation will show how by elevating the context of binary code, we can then apply higher-level analyses to address the more complex security issues introduced by C++ and COM. We describe how decompiling binary code into a high-level intermediate representation allows us to reconstruct high-level constructs like C++ objects and COM objects. With this high-level context regained, we can adapt and apply analyses that had only previously been applicable to source code analysis. Additionally, we can properly inform runtime enforcement engines to address security issues while still only being able to interface with applications through binary injection. We explain the feasibility of such work by suggesting the following thesis statement:

C++ and Microsoft's Component Object Model (COM) build upon the lower-level C *language. To allow for object oriented design, these programming paradigms extend, sometimes to a dangerous extent, the features of C, allowing developers to inadvertently introduce complex security vulnerabilities. The static analysis and runtime enforcement frameworks intended to address these types of vulnerabilities operate in a lower-level context than that in which the vulnerabilities were introduced. By reconstructing C++ and COM objects from their compiled counterpart, we can effectively elevate the context of binary-level frameworks, allowing for the development of more complex analyses and enforcement logic to address previously undetectable vulnerabilities.*

The security vulnerabilities presented in this dissertation continue to plague the software world. With the current state of affairs, we are attempting to solve extremely complex problems with comparatively simple solutions. Developments like those presented in this work are a necessary step toward increasing the security of applications that are critical to our everyday lives.

1.2 Contributions

In this dissertation, we make the following contributions:

- **Construct a general C++ decompilation framework:** We create a framework for reversing C++ compiled code into the intermediate representation used by the LLVM compiler infrastructure, allowing an analyst to employ any of the dozens of pre-built analyses that ship with LLVM. We implemented this system as a plugin for the popular IDA Pro disassembler. With this framework, we are able to resolve dynamically dispatched virtual calls in C++.

- **Develop an object reaching definition analysis algorithm for the detection of type confusion vulnerabilities:** Using the framework described above, we were able to implement a set of data flow algorithms which can trace object types from their instantiation to their use. Then, at each use point, we can programmatically determine whether the usage of that object conforms to its definition. By finding cases where objects are treated as an incorrect type, we can find instances of vtable escape vulnerabilities.

- **Develop a general-purpose data flow algorithm for detection of use-after-free conditions:** We present and formally define a data flow algorithm based on a technique from compiler theory known as available expression analysis. Our technique, called Available Object Definition Analysis (AODA) can be used by source code analysis tools, compilers and binary static analysis frameworks to identify use-after-free conditions.

- **Discover and characterize a systemic weakness in the COM security infrastructure**: We demonstrate that the existing killbit list security policies governing the instantiation of COM objects can easily be circumvented. We show that due to a weakness in the underlying COM architecture, many COM objects that are part of the default installation of the operating system can load other objects without ever consulting a policy of any kind. We demonstrate that this attack is possible through virtually every application that is commonly installed on Windows.

- **Design and implement a prototype policy enforcement infrastructure for COM objects:** We design and implement a prototype infrastructure for the system-wide COM instantiation policy enforcement, which we call COMBlocker. In this system, we are able to quickly compare (average lookup time of 554µs) the instantiation of COM objects against a global policy. Our approach is effective as it uses binary rewriting to force all COM object instantiations to be compared against the global policy, thus mitigating the above attack. We then compare our proposed solution to the patch recently issued by Microsoft (Security Bulletin MS10-036), which was created in response to our private disclosure of the vulnerability.

- **Discover and define several new classes of vulnerability:** In this work we provide the definition of several new exploitable vulnerability classes including C++ constructor failure, COM retention and initialization failure, COM type confusion and trust transitivity, and COM object release failures.

1.3 Dissertation Outline

In Chapter 2, we discuss the related research to this field. Here we will find that significant effort has been applied to the static analysis of source code, and the analysis of C-level security issues in binary code, but little work has been applied to the analysis of high-level constructs at the binary level.

Chapters 3 provides the background required for the reader to understand the low-level technical details of how these vulnerabilities arise and how they can be addressed. A through understanding of how a compiler translates high-level language constructs to a context that can be understood by a processor is necessary to understand how these issues can be detected through static analysis and enforced at runtime.

Chapters 4, 5, and 6 discuss the security issues that arise with complex types as they are instantiated, during their lifetime, and as they are deleted. Failures during object instantiation can result in very subtle bugs that can lead to exploitation. Throughout the lifetime of and object, there are several ways a developer can introduce security issues like type confusion. The improper deletion of objects can lead to use-after-free conditions.

Finally, Chapter 7 concludes the dissertation and discusses the future of research in this field.

Chapter II

RELATED WORK

2.1 Binary Analysis Frameworks

The focus of this work is on elevating the context of binary applications for the purposes of static analysis and runtime enforcement. Many tools have been developed in the past for the purposes of reversing binary code into an intermediate representation. Cousot and Cousot showed that by restructuring a language into an abstract representation, complex analyses are more easily implemented [19]. Following this concept, Song, et al. developed the BitBlaze framework for binary decompilation and analysis [81]. Several open source and commercial tools exist specifically for the decompilation of binary code. For example, the popular Hex-Rays plugin for IDA Pro reverses binary code to a C-like intermediate representation [39]. Dullien and Porst developed the Reverse Engineering Intermediate Language (REIL) for their commercial product, Bindiff [25].

None of these frameworks were appropriate for the goal of this work - focusing on security vulnerabilities introduced by higher-level language constructs. They all focus on reversing binary code to a C-like intermediate representation, thus leaving the IR at least one level lower than the higher-level frameworks in focus in this work. The LLVM IR natively supports the concept of object orientation. ReCall allows for the decompilation of objects into the LLVM IR format, making them available to all of the higher-level analyses we developed. Of course, this is not the only way to achieve higher-level context from binary code. Binary data structure recovery has been studied for use in host-based intrusion prevention systems, forensic analysis, and reverse engineering. For example, Dolan-Gavitt et al. [23] developed a dynamic-analysis system that creates attack detection signatures by monitoring kernel data structures in a way that is resistant to evasion. Similarly, Cozzie et al. developed Laika [20], a system that uses Bayesian unsupervised learning to detect the presence of data structures in memory indicative of a bot infection. Slowinska et al. [80] created a system that recovers data structures from a compiled binary for the purpose of reverse engineering.

This same problem exists for runtime analysis frameworks. Such tools either must be instrumented into the application at compile time or interface with the compiled application in its binary form. Because of this, the insertion points of the runtime framework have to operate at a contextual layer lower than the security issues discussed in this work. Here we present a runtime analysis framework that uses binary insertion to enforce security constraints on higher-level frameworks. Our runtime analysis framework, COMBlocker, allows for the dynamic enforcement of COM objects by inserting itself into compiled binaries.

2.2 Source Code Analysis Frameworks

Software vulnerability detection systems have existed for many years. Lint, created in 1977, has the ability to find flaws in the source code of C programs [46]. Lint-like systems have been developed over the years for the analysis of C programs. For example, Sparse [82] is a tool designed to find flaws in the Linux kernel. Splint [83] is the modern-day and maintained version of Lint, and Clang [15] is a popular compiler with built-in static analysis capability. Larochelle and Evans built upon these early works to statically detect the presence of buffer overflow vulnerabilities in source code [48, 27]. Shankar et al. developed a system to statically detect format string vulnerabilities [79]. ARCHER, a system developed by Xie, Chou, and Engler, uses a constraint solver to determine the safety of array accesses [88]. Similarly, Austin et al. created a system to detect pointer and array access errors [5]. Yet even with all the work in this area, Heelan points out that these problems are still unsolved [38].

As simple buffer overflows and format string vulnerabilities became increasingly rare, research focused on the detection of dynamic memory errors. Evans [29] and Bush et al. [10] present differing approaches to the detection of dynamic memory errors. These concepts were readily extended to the analysis of compiled C programs. Bugscam [9] is one of the oldest of these types of binary scanning tools and has been used to discover hundreds of vulnerabilities since its release. An entire industry has grown from these early tools: companies like Ounce Labs (now part of IBM), Coverity, Fortify Software (now part of HP), and Veracode all offer commercial products and services for the analysis of source code and compiled binaries.

Indeed static code analysis frameworks focused on higher-level language constructs already exist, but in order to maintain the needed context, they operate on source code - where context has not been lost by compilation. For example, research has been done in the area of C++ object reconstruction [77], which relies on access to the source code or runtime type information (RTTI). Additionally, Viega, et al. developed ITS4 [86]—a vulnerability scanner with support for C++. In this work, we reconstruct C++ objects from their compiled binary equivalent without access to any additional information. Many static analyses have been integrated directly into popular compilers to provide developers with warnings and errors as they are building their software. C++ analyses typically come in the form of type-checking and checks for const-ness and volatility and are fully enumerated in the C++ standard [75]. Significant research has attempted to extend required checks with virtual function call resolution in C++ programs. Bacon and Sweeny [6], for example, developed a static analysis algorithm to determine whether dynamic dispatch is truly necessary for a given method call. In cases where it is not, it can be replaced with a static function call, thus reducing the size of the compiled binary and the complexity of the program. Pande and Ryder [74, 73] and Calder and Grunwald [12] continue this concept to eliminate late binding where possible to take advantage of instruction pipelining on modern-day processors. SAFECode, a system developed by Dhurjati, Kowshik, and Adve [21] introduces a new type system that can be enforced at compile-time to prevent several types of vulnerabilities.

2.3 High-Level Analyses

The concept of analyzing code for higher-level security issues is not unique to this work. However, this work is unique in its application of high-level analyses to binary code, where it was previously believed too much context had been lost. The following subsections describe high-level analyses that are applied to source code.

2.3.1 Type Confusion Detection

SafeDispatch, developed by Jang, et al. analyzes source code for vtable escape vulnerabilities[44]. The GCC compiler has recently been extended by Google [36] to enable "vtable verification." Microsoft also added extensions to Visual Studio for the runtime detection of vtable escape vulnerabilities [67]. While all of these techniques are interesting, they only apply to new applications that are compiled with these tools enabled. They do not apply to the myriad applications that are already deployed that were built without these tools. Several tools have been developed to run during compilation to detect spatial safety issues [5, 47].

2.3.2 Reference Counting

Reference counting [17, 76, 50] is a method for tracking the lifetime of an object. Conceptually, it keeps a counter for every usage of the object. Each time an application needs the object, it increments the counter. When it is done with the object, it decrements the counter. Objects should not be deleted until the counter is zero. Automatic Reference Counting [16] is used extensively by Apple's Xcode development environment. In theory, reference counting prevents the deletion of objects that are still in use, and use-after-free conditions. However, as will be seen later in this dissertation, it is often implemented incorrectly resulting in exploitable security issues.

2.3.3 Memory Safety

Many tools have been developed for the enforcement of memory safety. Many of these tools insert additional checks into code before using memory-resident objects. Examples of such tools are CCured [70, 69], Cyclone [45], Purify [37], and Deputy [18]. These tools often require the developer to insert annotations in their code which can lead to missing vulnerabilities. Additionally, like above, they only apply to new applications that are developed with these toolsets. The work presented later in this dissertation shares many attributes of dangling pointer analysis. Dhurjati and Adve propose a solution by which they allocate an new virtual page of memory for each allocation [22]. In the attack that will be later described in Chapter 6, we describe how memory reuse in a very specific way is required for the exploitation of use-after-free conditions [78]. DieHard and DieHarder use randomization to reduce the likelihood that these precise requirements can be met [8, 72].

2.3.4 Control Flow Integrity

Control Flow Integrity (CFI) is one of the first areas that garnered attention from the research community. This was driven by the ease of exploitation and prevalence of stack overflow vulnerabilities. In short, the concept is to ensure that all code branches go to a proper location [1]. Zhang, et al. showed that CFI can be implemented with minimal runtime overhead [91]. Zeng, et. al showed how to introduce CFI using only static analysis [90]. Zhang and Sekar then later showed how CFI could be enforced with a low-level reference monitor [92]. Control Flow Integrity is an area of security that *is inline* with the low-level analysis frameworks that are already in existence. In this work, we focus on security issues that are introduced at much higher levels of context than CFI issues. However, in the face of all of these solutions, it has been shown that dangling pointer issues can be exploited to bypass CFI [14, 35].

2.3.5 Use-After-Free Detection

With binary code properly elevated to a higher-level context, we can now focus on higher-level vulnerabilities. Use-after-free vulnerabilities are an example of one such case. Searching specifically for use-after-free vulnerabilities, Caballero, et al. developed *Undangle* [11]; a run-time taint tracking tool used to detect dangling pointers. Also focused on dangling pointers, Lee, et al. created DangNull[49] to nullify class pointers when objects are deleted. While this is a sound solution to prevent exploitation of use-after-free vulnerabilities, it relies on the application's pre-existing ability to handle null pointers, else it would introduce stability problems [51]. Another example of C++specific vulnerabilities comes in the form of type confusion. Tice [85] proposed a compile time solution to verify the validity of a virtual function pointer before its invocation via inserting verification checks. Cling only allows for memory reuse by objects of the same type [3]. The closest work to what is found in this dissertation was done by Feist, et al. In their work [30], they use a memory model to detect use-after-free conditions in binary code. They use a memory model based approach because they focus on use-after-free conditions on primitive types.

2.4 COM Security

At yet an even higher level of context, we have Microsoft's Component Object Model (COM), and the ActiveX framework built on top of it. ActiveX has garnered significant attention from security experts over the past several years. Attacks against these controls range in severity from downloading files to an adversary-supplied location to arbitrary code execution [60, 61, 57, 58, 62]. As a variety of vulnerabilities have been found in COM objects included with the default installation of their operating systems, Microsoft has generally prevented the exploitation of the objects by adding them to a blacklist intended to prevent instantiation known as the killbit list. We previously demonstrated that it was possible to bypass the killbit settings in Internet Explorer using a vulnerability for ActiveX controls [24]. However, while this weakness was patched for Internet Explorer [63], the general susceptibility of the COM architecture and applications relying upon it has not previously been investigated.

The issue of securing the execution of content published by potentially untrusted third parties is not unique to COM or Windows. Java applets, for example, expose a web browser to similar classes of threats encountered by ActiveX [4, 53]. Security of Java applets has been studied extensively with several solutions proposed to verify the publisher of the content and enforce access rights on the content as it executes. Jaeger, et al. [43] and Islam, et al. [41] developed systems based on public key cryptography to verify the publisher of dynamically downloadable executable content. They both then go on to propose solutions for the enforcement of access controls on the code as it executes.

Security policies have been developed to address whether COM objects should be loaded by Internet Explorer. The killbit list attempts to prevent the instantiation of known bad controls [59]. As documented by Loscocco et al., ActiveX controls can be signed similar to the way that was proposed for Java [52]. Additionally, ActiveX supports the concept of "Safe for Scripting" [54]. This allows a control to tell Internet Explorer whether it can be safely loaded by the script engine. Microsoft has implemented additional security policies governing the instantiation of COM objects in other applications including the MS Office suite [66]. However, policies governing the instantiation of COM objects are implemented by the COM container itself.

Security retrofits to well-established software infrastructures such as COM generally require significant effort. A number of researchers have investigated the problem of retrofitting legacy systems with security infrastructure. These efforts use a range of approaches including static analysis of code [13, 93, 71, 89, 31] and the monitoring of program behavior to determine security sensitive operations [42, 34]. The work by Ganapathy, et al. demonstrates the injection of authorization policy enforcement code into existing legacy applications including web and proxy servers [33] and the Linux Security Module infrastructure [32]. Fraser, et al. perform a similar deployment of authorization hooks in MINIX [31]. This builds on the concept of inlined reference monitoring as described by Bauer, et al. [7], Erlingsson [26], and Evans and Twyman [28]. Each of these demonstrates how an existing application can be modified to perform functions not originally designed into the code. The closed nature of the COM architecture requires that our solution rely on binary rewriting techniques rather than those used on source code.

Each of the vulnerability classes discussed in this work is introduced into an application from a higher-level language construct or framework. The elevation of context from the binary level to one commensurate with the language or framework in question is required to properly analyze applications for security issues and enforce their behavior at runtime.

Chapter III

BACKGROUND

3.1 Complex Types Built on a Primitive Language

The security issues described throughout this dissertation all have one thing in common — they are issues present in higher-level language constructs that are built on top of a primitive language (in this case, C). In many ways, the overlay of these complex concepts on top of a primitive language is *the cause* of the security issues. By building complex datatypes on top of a primitive language, there is no opportunity for type verification which leads to problems like type confusion as described in Chapter 5. Since C does not natively provide for reference counting of variables, the only method for self-policing the allocation of memory is by tracking scope. This is insufficient for object-oriented programming where objects most often must live beyond the scope of a single function. With that, memory allocation and reference counting are implemented on top of the primitive language and allow for opportunities to make mistakes. This leads to conditions like memory leaks, use-after-free conditions, and null pointer dereferences. To better understand the low-level details of the complex types we will be exploring, the following section cover how objects are created, used, and deleted in C++, and how COM objects span the same lifecycle.

3.2 C++ Background

3.2.1 Defining a Class

C++ builds on the C `struct` data type. It allows developers to create class definitions by adding additional functionality to what is traditionally offered in a `struct`. A short summary of the additions to the C `struct` are enumerated below:

1. **Namespaces** - Namespaces allow a developer to avoid naming collisions by specifying a higher-level scope for each variable declaration. Namespaces can be defined by using the `namespace` keyword, but are also implicitly defined by a class definition. For example consider the code below. Here we can see three declarations of

the variable `referenceCount` and two declarations of the type `Class1`. Namespaces allow the developer to disambiguate between all of these definitions. In this example, `example::Class1` is different than `Class1`, and `example::Class1::referenceCount`, `Class1::referenceCount`, and `referenceCount` are all different from one another.

```
namespace example {
        class Class1 {
                int referenceCount;
        };
}

class Class1 {
        int referenceCount;
};

int referenceCount;
```

Figure 1: Example use of namespaces

2. **Constructors -** A constructor is a function that is automatically called when an object is instantiated. The constructor is defined by creating a function of the same name of the class type. For example, consider the code below. Here we can see and object of type `Class1` the constructor is defined as the method named `Class1::Class1()`. This is where the developer would insert code required to properly initialize an object.

```
class Class1 {
        Class1();
};

Class1::Class1()
{
        //do all of the object initialization here
}
```

Figure 2: Example constructor declaration

3. **Destructors -** Destructors are the counterpart to constructors. This is where a developer would insert code that is necessary to clean up just prior to an object being deleted. Destructors are specified by declaring an object method with the same name as the class and prefacing the name with a tilde (~). An example of a destructor can be seen in the code below.

```
class Class1 {
        Class1();
        ~Class1();
};

Class1::Class1()
{
        //do all of the object initialization here
}

Class1::~Class1()
{
        //do any object clean up here
}
```

Figure 3: Example destructor declaration

4. **Inheritance** - Inheritance allows a class to adopt the properties and methods of a parent class. In the code below, the developer has declared a class of type `Class1`. Then, a second class of type `Class2` has been declared that inherits from `Class1`. With this inheritance, `Class2` will have a class method named `Method1()` in addition to the method `Method2()` that is specified in the `Class2` definition.

```
class Class1 {
        Class1();
        Method1();
};

class Class2 : Class1 {
        Class2();
        Method2();
};
```

Figure 4: Example of inheritance

5. **`public` and `private` Operators -** C++ offers the `public` and `private` keywords in class definitions. These keywords declare which properties and methods are accessible from anywhere in the program, and those that are only accessible from within the class itself. In the code example below, `Class1` has a public method named `PublicMethod()` and a public property called `publicProperty`. In these cases, from anywhere in a program, a developer can directly call `PublicMethod()` on an instance of the object. Additionally, the developer could directly access the property `publicProperty`. The code in Figure 6 shows a case where a developer has instantiated an object of type `Class1` called `my_class1`. Here we can see a call to the public method and an access to the public property. On the contrary, the code in Figure 7 shows similar code where the develop attempts to access a private method and private property. This code will fail compilation with an error. The code in Figure 8 shows how the private methods and properties can be accessed from within the class itself. It is important to note that

the enforcement of public and private is a compile-time check. There is no runtime enforcement of these modifiers.

```
class Class1 {
        Class1();
public:
        PublicMethod();
private:
        PrivateMethod();
public:
        int publicProperty;
private:
        int privateProperty;
};
```

Figure 5: Examples of use of public and private keywords

```
int main(int argc, char **argv)
{
        Class1 my_class1;

        my_class1.PublicMethod();

        int local_var = my_class1.publicProperty;
}
```

Figure 6: Examples of accessing public properties and methods

```
int main(char **argv, int argc)
{
        Class1 my_class1;

        my_class1.PrivateMethod();

        int local_var = my_class1.privateProperty;
}
```

Figure 7: Examples of attempting to access private methods and properties. This code will fail compilation with an error.

```
class Class1 {
        Class1();
public:
        PublicMethod()
        {
                PrivateMethod();
                privateProperty = 1;
        };
private:
        PrivateMethod();
public:
        int publicProperty;
private:
        int privateProperty;
};
```

Figure 8: Examples of properly accessing private methods and properties.

6. `virtual` **Operator** - The `virtual` keyword is what declares a class method as being overridable. In the code below we see a case where a class of type `Class2` inherits from `Class1`. However, `Class2` overrides the definition of `Method2()` with its own

definition.

```
class Class1 {
        Class1();
        int Method1();
        virtual int Method2();
};

class Class2 : Class1 {
        Class2();
        int Method2();
};
```

Figure 9: Examples of virtual methods

3.2.2 Object Lifetime

3.2.2.1 Creating an Object

There are two ways for a developer to instantiate an object. The first is to declare the object as a stack variable just as any other type of variable. For example, the code in Figure 10 shows three variables declared on the stack: an integer, a character array length 32, and an object of type `class1`. For this code to be valid, `class1` would have to be defined as shown in Figure 2.

```
int main(int argc, char **argv)
{
        int i;
        char some_string[32];
        class1 my_class;

        . . .
}
```

Figure 10: Example of stack-declared variables

When an object is declared on the stack, the class constructor is called immediately upon declaration.

The second way to instantiate an object is to declare the object on the heap. To do this, a developer uses the `new` operator. Behind the scenes, `new` automatically allocates enough memory to store the object. The exact structure of what is contained in memory is covered in detail in Section 3.2.3. The implied memory allocation occurs through a call to a compiler-inserted function called `YAPAXI()`. This is basically just a wrapper for `malloc()`. After allocating the proper amount of memory, `new` automatically calls the object's constructor passing the pointer returned from `YAPAXI()` as the `this` pointer. The return from the constructor is the same `this` pointer, but now it points to a complete object. Much like the use of `malloc()`, it is up to the developer to manually free the memory used by the object after it is no longer in use. There is no implicit deletion like in the case of a stack-declared object.

3.2.2.2 Deleting an Object

In the case of a stack-declared object, the object is automatically deleted, and the destructor is implicitly called when the variable goes out of scope. In the case of a heap-declared object, the developer must manually delete the object when it is no longer needed. This is done by using the `delete` operator. Behind the scenes, the compiler inserts a call to the object's destructor and frees the allocated memory. To free the memory, the compiler inserts a call to a function called `YAXPAX()`. This is basically just a wrapper for `free()`. Because freeing the object is at the discretion of the developer, one can envision how memory leaks and use-after-free conditions may occur. Use-after-free conditions are covered in detail in Chapter 6.

3.2.3 Structure of an Object After Compilation

An important distinction is how methods are inserted into the code when they are declared `virtual` and when they are not. When a method is not declared virtual, calls to that method are dispatched exactly the same way that function calls are made in C. For example, consider the code shown in Figure 11a. In the corresponding Figure, 11b, we can see the compiled

```
class class1 {
public:
    class1();
    ˜class1();
    void addRef();
    void print();
    virtual void voidFunc1()  ;
    virtual void debug();

private:
    int refCount;
};

int _tmain(int argc, _TCHAR* argv[])
{
    class1 C1;
    class2 C2;

    C1.print();
    C2.print();

    return 0;
}
```

(a) Original source

```
.text:00401120 wmain           proc near
.text:00401120
.text:00401120 var_1C          = byte ptr -1Ch
.text:00401120 var_14          = dword ptr -14h
.text:00401120 var_10          = byte ptr -10h
.text:00401120 var_C           = dword ptr -0Ch
.text:00401120 var_4           = dword ptr -4
.text:00401120
.text:00401120                 push    ebp
.text:00401121                 mov     ebp, esp
.text:00401123                 push    0FFFFFFFFh
.text:00401125                 push    offset loc_401B30
.text:0040112A                 mov     eax, large fs:0
.text:00401130                 push    eax
.text:00401131                 sub     esp, 10h
.text:00401134                 mov     eax, __security_cookie
.text:00401139                 xor     eax, ebp
.text:0040113B                 push    eax
.text:0040113C                 lea     eax, [ebp+var_C]
.text:0040113F                 mov     large fs:0, eax
.text:00401145                 lea     ecx, [ebp+var_1C]
.text:00401148                 call    class1__class1
.text:0040114D                 mov     [ebp+var_4], 0
.text:00401154                 lea     ecx, [ebp+var_10]
.text:00401157                 call    class2__class2
.text:0040115C                 mov     byte ptr [ebp+var_4], 1
.text:00401160                 lea     ecx, [ebp+var_1C]
.text:00401163                 call    class1__print
.text:00401168                 lea     ecx, [ebp+var_10]
.text:0040116B                 call    class2__print
.text:00401170                 mov     [ebp+var_14], 0
.text:00401177                 mov     byte ptr [ebp+var_4], 0
.text:0040117B                 lea     ecx, [ebp+var_10]
.text:0040117E                 call    class2___class2
.text:00401183                 mov     [ebp+var_4], 0FFFFFFFFh
.text:0040118A                 lea     ecx, [ebp+var_1C]
.text:0040118D                 call    class1___class1
.text:00401192                 mov     eax, [ebp+var_14]
.text:00401195                 mov     ecx, [ebp+var_C]
.text:00401198                 mov     large fs:0, ecx
.text:0040119F                 pop     ecx
.text:004011A0                 mov     esp, ebp
.text:004011A2                 pop     ebp
.text:004011A3                 retn
.text:004011A3 wmain           endp
```

(b) Corresponding binary

Figure 11: Example use of non-virtual class methods

version of the code. In the source code, we can see a call to the class method `print()` at line 18, we can see this is simply translated into a normal call in the compiled code (the call is located at line 0x401163). The only difference between this call, and a standard C call is the calling convention. Specifically, in C++, the `this` pointer for the object is automatically passed to the method in the `ecx` register.

In the case of methods that are declared `virtual`, a table of function pointers is created. One function pointer is included for each `virtual` function declared in the class. This table – known as the vtable – contains a function pointer for each `virtual` function in the order in which they are declared. A pointer to the vtable is stored as the first element in the object's structure in memory. This layout can been seen in more detail in Figure 12. When a call is made to a `virtual` function, the vtable is consulted to find the location of the associated function. It is this method of dynamic dispatch that allows classes to override method definitions of parent classes. When a class overrides a method, the compiler inserts the child class's definition of the method into the vtable in place of the parent class's definition.

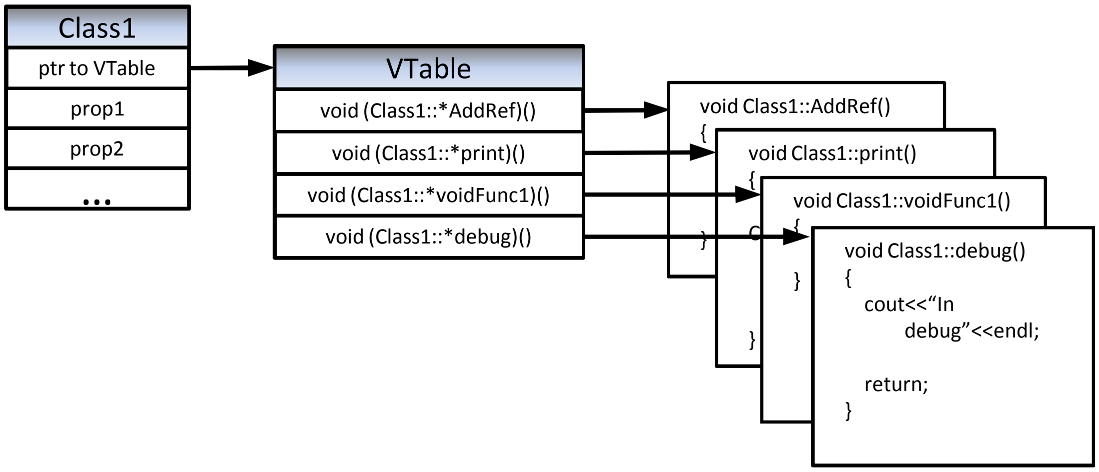


Figure 12: Memory layout of a C++ object after compilation

Consider the code in Figure 13a. This is very similar to the code we used in the non-virtual example. However, in this case, a call is made to the virtual method `debug()`. In the corresponding binary in Figure 13b, we can see a call to a dereferenced pointer at line 0x401179. This where the code is referencing the vtable for `class1` to find the pointer to the `debug()` method.

3.3 COM Background

3.3.1 COM Overview

COM is an architectural standard that mandates a language agnostic representation of objects, and facilitates interaction between these objects. Microsoft uses COM as a fundamental building block in many of their premier technologies. It is pervasive in their flagship Windows Operating System, and also utilized extensively by many other peripheral products, such as Internet Explorer and Office. In the first section entitled Variants, we will discuss the fundamental, language agnostic data types that COM uses to communicate and the APIs used to manipulate them. Variants will be explored in order to provide the reader with more context for the types of vulnerabilities that are the focus of this work. Following variants is a section entitled COM Automation, which discusses the subset of COM objects that can be readily exposed to scripting runtime environments, collectively known as ActiveX controls. Finally, in the section entitled COM Persistence Overview, we will discuss

```
class class1 {
public:
    class1();
    ~class1();
    void addRef();
    void print();
    virtual void voidFunc1()  ;
    virtual void debug();

private:
    int refCount;
};

int _tmain(int argc, _TCHAR* argv[])
{
    class1 C1;
    class2 C2;

    C1.print();
    C2.print();
    C1.debug();

    return 0;
}
```

(a) Original source

```
.text:00401120 wmain           proc near
.text:00401120
.text:00401120 var_20          = byte ptr -20h
.text:00401120 var_18          = dword ptr -18h
.text:00401120 var_14          = dword ptr -14h
.text:00401120 var_10          = byte ptr -10h
.text:00401120 var_C           = dword ptr -0Ch
.text:00401120 var_4           = dword ptr -4
.text:00401120
.text:00401120                 push    ebp
.text:00401121                 mov     ebp, esp
.text:00401123                 push    0FFFFFFFFh
.text:00401125                 push    offset loc_401B40
.text:0040112A                 mov     eax, large fs:0
.text:00401130                 push    eax
.text:00401131                 sub     esp, 14h
.text:00401134                 mov     eax, __security_cookie
.text:00401139                 xor     eax, ebp
.text:0040113B                 push    eax
.text:0040113C                 lea     eax, [ebp+var_C]
.text:0040113F                 mov     large fs:0, eax
.text:00401145                 lea     ecx, [ebp+var_20]
.text:00401148                 call    class1__class1
.text:0040114D                 mov     [ebp+var_4], 0
.text:00401154                 lea     ecx, [ebp+var_10]
.text:00401157                 call    class2__class2
.text:0040115C                 mov     byte ptr [ebp+var_4], 1
.text:00401160                 lea     eax, [ebp+var_10]
.text:00401163                 mov     [ebp+var_18], eax
.text:00401166                 lea     ecx, [ebp+var_20]
.text:00401169                 call    class1__print
.text:0040116E                 lea     ecx, [ebp+var_10]
.text:00401171                 call    class2__print
.text:00401173                 lea     eax, [ebp+var_20]
.text:00401176                 lea     ecx, [ebp+var_20]
.text:00401179                 call    dword ptr [eax+0ch]
.text:0040117E                 mov     [ebp+var_14], 0
.text:00401185                 mov     byte ptr [ebp+var_4], 0
.text:00401189                 lea     ecx, [ebp+var_10]
.text:0040118C                 call    class2___class2
.text:00401191                 mov     [ebp+var_4], 0FFFFFFFFh
.text:00401198                 lea     ecx, [ebp+var_20]
.text:0040119B                 call    class1___class1
.text:004011A0                 mov     eax, [ebp+var_14]
.text:004011A3                 mov     ecx, [ebp+var_C]
.text:004011A6                 mov     large fs:0, ecx
.text:004011AD                 pop     ecx
.text:004011AE                 mov     esp, ebp
.text:004011B0                 pop     ebp
.text:004011B1                 retn
.text:004011B1 wmain           endp
```

(b) Corresponding binary

Figure 13: Example use of virtual class methods

```
struct __tagVARIANT {
        VARTYPE vt;
        WORD wReserved1;
        WORD wReserved2;
        WORD wReserved3;
        union {
                BYTE bVal;
                SHORT iVal;
                FLOAT fltVal;
                DOUBLE dblVal;
                VARIANT_BOOL boolVal;

                ... more elements ...

                BSTR bstrVal;
                IUnknown *punkVal;
                IDispatch *pdispVal;
                SAFEARRAY *parray;
                VARIANT_BOOL *pboolVal;
                _VARIANT_BOOL *pbool;
                SCODE *pscode;
                CY *pcyVal;
                DATE *pdate;
                BSTR *pbstrVal;
                VARIANT *pvarVal;
                PVOID byref;
        } __VARIANT_NAME_1;
};
```

Figure 14: Type definition for `__tagVARIANT`

the concept of persistence - the ability to serialize the current state of a COM object and subsequently resurrect that object at a later time. The use of persistence will be explored in the context of potentially hostile environments, where the serialized objects may originate from untrusted sources (such as malicious web pages or office documents).

3.3.1.1 Variants

`VARIANT`s are one of the key data structures utilized throughout the Windows platform for representing arbitrary data types in a standardized format. In particular, they are an integral part of COM, and are employed to exchange data between two or more communicating objects. The `VARIANT` data structure is a relatively simple one – it is composed of a type and a value, and is defined in OAIdl.h in the Windows SDK as shown in Figure 14.

The value contained by a `VARIANT` can be one of a variety of different types, and so only has meaning when given context by the `vt` member, which indicates the type. There are

Table 1: `VARIANT` Basic Types

Type Name	Value	Union Contains
`VT_EMPTY`	0x0000	Undefined
`VT_NULL`	0x0001	NULL value
`VT_I2`	0x0002	Signed (2-byte) short
`VT_I4`	0x0003	Signed (4-byte) integer
`VT_R4`	0x0004	Signed (4-byte) real (float)
`VT_R8`	0x0005	Signed large (8-byte) real (double)
`VT_BSTR`	0x0008	String; Pointer to a `BSTR`
`VT_DISPATCH`	0x0009	Pointer to an `IDispatch` interface (automation object)
`VT_ERROR`	0x000A	Error code (4-byte integer)
`VT_BOOL`	0x000B	Boolean (2-byte short)
`VT_VARIANT`	0x000C	Pointer to another `VARIANT`
`VT_UNKNOWN`	0x000D	Pointer to an `IUnknown` interface (any COM object)
`VT_I1`	0x0010	Signed (1-byte) char
`VT_UI1`	0x0011	Unsigned (1-byte) char
`VT_UI2`	0x0012	Unsigned (2-byte) short
`VT_UI4`	0x0013	Unsigned (4-byte) integer
`VT_RECORD`	0x0024	Pointer to an `IRecordInfo` interface (used to represent user-defined data types)

quite a large number of basic types that can be represented by a `VARIANT`. Some of the more common ones are shown in Table **??**.

As can be seen in Table **??**, all of the basic data types can be represented as a variant, in addition to a variety of COM interface types such as `IUnknown` and `IDispatch` interfaces. Furthermore, user-defined types are supported through the use of the `IRecordInfo` COM interface. This interface provides functions to define custom object sizes and marshallers so that any arbitrary data structure can be represented. The listed types are only a subset of all the supported `VARIANT` types. A complete list of all of the available types can be found in wtypes.h located within the Windows SDK. In addition to basic variant types, there are several modifiers that, when used in conjunction with a basic type, alter the meaning of what is contained within the `__VARIANT_NAME_1` union. Modifiers cannot be used on their own; they are specifically designed to provide additional context to a basic type. They are used by combining the modifier value (or values) with that of the basic type. The modifiers and their respective meanings are summarized in Table 2.

Table 2: VARIANT Modifier Types

Modifier Name	Modifier Value	Value
VT_VECTOR	0x1000	Value points to a simple counted array (Rarely used)
VT_ARRAY	0x2000	Value points to a SAFEARRAY structure
VT_BYREF	0x4000	Value points to base type, instead of containing a literal of the base type

As can be seen in the tables, basic types are all below 0x0FFF, and modifiers are single-bit values larger than 0x0FFF. So, by augmenting a basic type with a modifier using simple bit-masking operations, a new, complex type is formed. For example, a VARIANT containing an array of strings would have the type VT_ARRAY|VT_BSTR, and the value member would point to a SAFEARRAY where each member was a BSTR. (SAFEARRAYs will be examined in more depth momentarily.) A VARIANT could represent a pointer to a signed integer by having the type VT_BYREF|VT_I4. The VT_BYREF modifier may also be used in conjunction with one of the other modifiers, so a VARIANT could have the type (VT_BYREF|VT_ARRAY|VT_BSTR). In this case, the value member would point to a SAFEARRAY pointer, whose members are all of type BSTR.

3.3.1.2 Safe Arrays

Arrays are a common data construct utilized by COM, and are present in VARIANTs that contain the VT_ARRAY modifier in the vt field. In this case, a SAFEARRAY is used to encapsulate a series of elements of the same data type, and can be manipulated through the SafeArray API for safely accessing the members of the array without needing to worry about boundaries and other administrative problems associated with array access. Although they are most often used to represent an array with just a single dimension, SAFEARRAYs are also capable of representing mulch-dimensional arrays of potentially differing dimension sizes (often referred to as “jagged arrays”). The SAFEARRAY structure definition is defined in OAIdl.h in the Windows SDK, and is shown below.

Elements contained within a SAFEARRAY are cbElements in size, and are stored contiguously in an area of memory, pointed to by the pvData member. An array of SAFEARRAYBOUND structures follows the SAFEARRAY descriptor in memory, with each SAFEARRAYBOUND structure

```
typedef struct tagSAFEARRAY {
        USHORT cDims;
        USHORT fFeatures;
        ULONG cbElements;
        ULONG cLocks;
        PVOID pvData;
        SAFEARRAYBOUND rgsabound[ 1 ];
} SAFEARRAY;
```

Figure 15: Type definition for `tagSAFEARRAY`

```
typedef struct tagSAFEARRAYBOUND {
        ULONG cElements;
        LONG lLbound;
} SAFEARRAYBOUND;
```

Figure 16: Type definition for `tagSAFEARRAYBOUND`

describing a single dimension of the array. The `SAFEARRAYBOUND` structure is constructed as follows:

Simply put, the lLbound member indicates the lower bound of the described dimension, and the `cElements` member indicates how many members exist within that dimension. The `SAFEARRAY` API is relatively extensive, so we will consider the most common API functions required for manipulation of these structures. The first two functions are for initialization and destruction, and are the complement of each other:

```
SAFEARRAY *SafeArrayCreate(VARTYPE vt,
         UINT cDims, SAFEARRAYBOUND * rgsabound);
```

```
HRESULT SafeArrayDestroy(SAFEARRAY * psa);
```

These functions are used to create and destroy an array respectively. When the array is created, the data type of each array member is designated, as well as the number of the dimensions of the array. These properties are both immutable; a `SAFEARRAY`'s type and number of dimensions cannot be modified after creation. There are two different ways of accessing data in arrays. The first way is to get a pointer to the memory where all of the elements reside, and is done using the following functions:

```
HRESULT SafeArrayAccessData(SAFEARRAY * psa,
        void HUGEP** ppvData);

HRESULT SafeArrayUnaccessData(SAFEARRAY * psa);
```

This is often the preferred method when accessing elements in a loop, in the form:

```
BSTR *pString;

if(FAILED(SafeArrayAccessData(psa, &pString))
        return -1;

for(i = 0; i < psa->rgsabound[0].cElements; i++) {
        ... operate on string ...
}

SafeArrayUnaccessData(psa);
```

The second way to access data is by accessing an individual element using the following functions:

```
SafeArrayGetElement(SAFEARRAY * psa, LONG * rgIndices,
        void * pv);

SafeArrayPutElement(SAFEARRAY * psa, LONG * rgIndices,
        void * pv);
```

Each of these functions takes an array of indices and will either return or store the specific value in question. Note that internally, both functions verify the validity of the supplied indices to ensure that each array access is within bounds. Lastly, we should mention that SAFEARRAYs have locking mechanisms to ensure exclusive thread access to array data, accessed by the following two functions:

```
HRESULT SafeArrayLock(SAFEARRAY * psa);

HRESULT SafeArrayUnlock(SAFEARRAY * psa);
```

3.3.1.3 VARIANT versus VARIANTARG

Many of the `VARIANT` API functions take either a `VARIANT` or a `VARIANTARG`. Microsoft documentation suggests that the difference between these two values is that `VARIANT`s always contain direct values (i.e., they can't have the modifier `VT_BYREF`), whereas `VARIANTARG`s can. In fact, you will notice in the discussion of the `VARIANT` API further on that most of the Variant* functions take `VARIANTARG`s. In reality, these structures are actually equivalent and can be used interchangeably despite documentation indicating otherwise. Furthermore, a compiler error is not generated when they are used interchangeably.

3.3.1.4 `VARIANT` *API*

The API for manipulating `VARIANT`s is quite extensive, however only a few of the functions are relevant for the purposes of this dissertation, and they are discussed in this section.

3.3.1.5 Variant Initialization and Destruction

`VARIANT`s are initialized using the `VariantInit()` function, which has the following prototype:

```
HRESULT VariantInit(VARIANTARG *pvarg);
```

This function does nothing except to set the type member of the `VARIANT`, `vt`, to `VT_EMPTY`, indicating that the `VARIANT` holds no value. The `VARIANT` is later cleaned up using the reciprocal function `VariantClear()`:

```
HRESULT VariantClear(VARIANTARG *pvarg);
```

The `VariantClear()` function will also clear the vt member, as well as free any data associated with the `VARIANT`. For example, if the `VARIANT` contains an `IDispatch` or `IUnknown` interface (type `VT_DISPATCH` or `VT_UNKNOWN` respectively), then the interface will be released by `VariantClear()`. If the `VARIANT` is a string (`VT_BSTR`), it will be de-allocated, and so on.

3.3.1.6 Variant Manipulation

The two primary types of operations one may perform on a `VARIANT` using the API are conversion and duplication. There are a large variety of specific conversion functions of the

```
HRESULT VariantCopy(VARIANTARG *pvargDest,
        VARIANTARG *pvargSrc);

HRESULT VariantCopyInd(VARIANTARG *pvargDest,
        VARIANTARG *pvargSrc);
```

form VarXXFromYY(), where XX is the destination VARIANT type and YY is the source type. There are also generic functions for converting between any two VARIANT types, which are shown below.

```
HRESULT VariantChangeType(VARIANTARG *pvargDest,
        VARIANTARG *pvargSrc, unsigned short wFlags, VARTYPE vt);

HRESULT VariantChangeTypeEx(VARIANTARG *pvargDest,
        VARIANTARG *pvargSrc, LCID lcid,
        unsigned short wFlags, VARTYPE vt);
```

These two functions both perform essentially the same task – converting pvargSrc to the type specified by vt, and placing the result in pvargDest. The other functions worth mentioning are those responsible for copying a VARIANT value from one VARIANT to another:

These functions both clear the destination VARIANT, and then copy in the source VARIANT. They do a deep copy; that is, if a COM interface is copied, the reference count is incremented, and so on. The difference between the two functions is that VariantCopyInd() will follow an indirect reference for a copy (i.e., if the VARIANT has the VT_BYREF modifier, the value will be dereferenced and then modified), whereas VariantCopy() will not. VariantCopyInd() is also recursive; if a VARIANT is received that has the type (VT_BYREF|VT_VARIANT), the destination VARIANT will be examined further. If it is also a (VT_BYREF|VT_VARIANT), an error is signaled. If it has a VT_BYREF modifier but is not a VT_VARIANT, this VARIANT will be passed to VariantCopyInd() again, thus retrieving the value being stored.

3.3.1.7 COM Automation

As mentioned previously, COM Automation facilitates the integration of pluggable components into scripting environments. This is primarily achieved by creating objects that

```
/*** IDispatch methods ***/
HRESULT (STDMETHODCALLTYPE *GetTypeInfoCount)( IDispatch* This,
        UINT* pctinfo);
HRESULT (STDMETHODCALLTYPE *GetTypeInfo)( IDispatch* This, UINT iTInfo,
        LCID lcid, ITypeInfo** ppTInfo);
HRESULT (STDMETHODCALLTYPE *GetIDsOfNames)( IDispatch* This, REFIID riid,
        LPOLESTR* rgszNames, UINT cNames, LCID lcid, DISPID* rgDispId);
HRESULT (STDMETHODCALLTYPE *Invoke)( IDispatch* This, DISPID dispIdMember,
        REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS* pDispParams,
        VARIANT* pVarResult, EXCEPINFO* pExcepInfo, UINT* puArgErr);
```

Figure 17: List of `IDispatch` methods

implement one or both of the automation interfaces: `IDispatch` and `IDispatchEx`. The `IDispatch` interface exposes functions that are designed to achieve the following directives:

1. Allow an object to be self-publishing (i.e., advertise its properties and methods)

2. Allow methods to be called or properties to be manipulated by name, rather than direct vtable / memory manipulation.

3. Provide a unified marshaling interface for objects being passed to methods or properties, as well as objects being returned to the scripting host.

By implementing `IDispatch`, objects can be loaded at runtime by a host application and subsequently manipulated without the host having to know any compile time details about the objects it is. This capability is particularly useful for scripting interfaces which require extensibility. The `IDispatch` interface is derived from `IUnknown` (both documented at MSDN), adding four methods as shown:

If an application would like to call any of the methods or modify any of the properties exposed by the object, it first needs to determine the dispatch ID associated with the method it would like to call. To determine this information, the application first needs to call `GetIdsOfNames()`. The return value is an integer that maps to the actual method that will be executed through the `Invoke()` method. The `Invoke()` method takes the ID of the member to be executed, the arguments to the method, and some other information about locale, etc as arguments. The `wFlags` argument passed to `Invoke()` defines whether the dispatch ID references a method exposed by the object or a property value that it should

```
typedef struct FARSTRUCT tagDISPPARAMS{
        VARIANTARG FAR* rgvarg; // Array of arguments.
        DISPID FAR* rgdispidNamedArgs; // Dispatch IDs of named arguments.
        unsigned int cArgs; // Number of arguments.
        unsigned int cNamedArgs; // Number of named arguments.
} DISPPARAMS;
```

Figure 18: Type definition for `tagDISPPARAMS`

either get or set. The arguments to the method that will be executed are passed in a `DISPPARAMS` structure. The `DISPPARAMS` structure is defined below:

As you can see, this structure passes the arguments to the method in an array of `VARIANT`s (See the section on `VARIANT`s for more detail). This array must be unmarshalled by the called method. In some cases, this can be a bit of a daunting task given the complexity of some of the `VARIANT` types that may be present in the array. The `IDispatch` interface is useful for creating automation objects whose behavior is immutable - the properties and methods must be known at compile time and they don't change. However, in some cases, it is desirable to have objects whose behavior could be modified at runtime, and the `IDispatchEx` interface extends `IDispatch` to allow this additional functionality. With `IDispatchEx` objects, it is possible to add or remove properties or methods at runtime. This is functionality that is commonly required by more dynamic late-bound languages such as scripting languages (e.g. JavaScript). The `IDispatchEx` interface is also derived from the `IUnknown` interface, adding eight methods as follows:

```
HRESULT DeleteMemberByDispID( DISPID id );

HRESULT DeleteMemberByName( BSTR bstrName,
        DWORD grfdex );

HRESULT GetDispID( BSTR bstrName,
        DWORD grfdex, DISPID *pid );

HRESULT GetMemberName( DISPID id,
        BSTR *pbstrName );
```

```
HRESULT GetMemberProperties( DISPID id,
        DWORD grfdexFetch, DWORD *pgrfdex );

HRESULT GetNameSpaceParent( IUnknown **ppunk );

HRESULT GetNextDispID( DWORD grfdex,
        DISPID id, DISPID *pid );

HRESULT InvokeEx( DISPID id, LCID lcid,
        WORD wFlags, DISPARAMS *pdp,
        VARIANT *pVarRes, EXCEPINFO *pei,
        IServiceProvider *pspCaller );
```

While there are some differences in the way dispatch IDs are retrieved, the main changes to `IDispatchEx` are those that allow for the creation and deletion of object properties and methods. `GetDispID()`, for example, differs from `GetIdsOfNames()` in that it can be told to create a new name and dispatch ID for a new property or method. Additionally, you can see the methods `DeleteMemberByName()` and `DeleteMemberByDispID()` have been added. In ActiveX controls that extend the `IDispatchEx` interface, the dynamic creation and deletion of members is accessible through JavaScript. Interestingly, JavaScript (for Internet Explorer) itself is implemented using a modified `IDispatchEx` interface exposed by the Microsoft script engine. Conceptually, this implementation makes sense because JavaScript will need to be able to create objects and add and delete members all without any preconceived notion of what the object may look like. So, for example, when JavaScript creates a new object:

```
obj = new Object();
```

Internet Explorer will first call the `GetDispID()` method for Obj – ensuring the `fdexNameEnsure` flag is set to create the member. It will then call its own internal version of `Invoke()` to call the `Object()` method. The value returned from the call to `Invoke()` will then become assigned to the `obj` member.

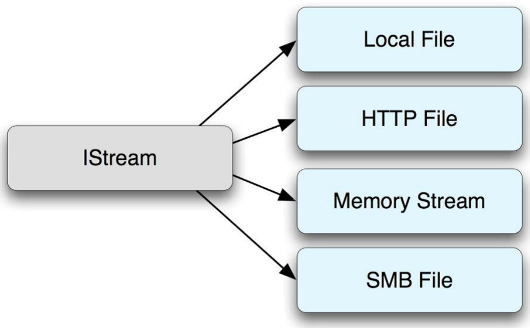


Figure 19: Diagram representing various media that can contain the `IStream` data.

3.3.1.8 COM Persistence Overview

COM provides two primary interfaces for manipulating an object's persistence data. The first interface, `IStream`, represents a data stream that is used to store a single object's persisted data. It supports standard file operations including reading, writing, and seeking using the interface methods. The `IStream` interface abstracts the underlying storage details from the consumer of the stream. This abstraction allows for COM objects to implement serialization functionality without explicit knowledge of the underlying backing store. This abstraction is visually depicted in Figure 19.

The second interface, `IStorage`, is employed when a program or COM object requires the persistence of multiple objects. `IStorage` represents a storage file, which can hold logically separate binary streams inside a single file using unique names to identify each stream. Additionally, a storage file can contain logically separate subordinate storage files, also accessed by unique names, thus allowing for recursion if it is required. The `IStorage` interface provides methods that allow the programmer to access each of the constituent streams and subordinate storage files. Figure 20 depicts an example of a typical storage file.

In addition to `IStream` and `IStorage`, there are several other interfaces that can be used for manipulating COM persistence data, depending on the medium that contains the data. The following is a list of interfaces that can store persistent object data:

- `IMoniker`
- `IFile`

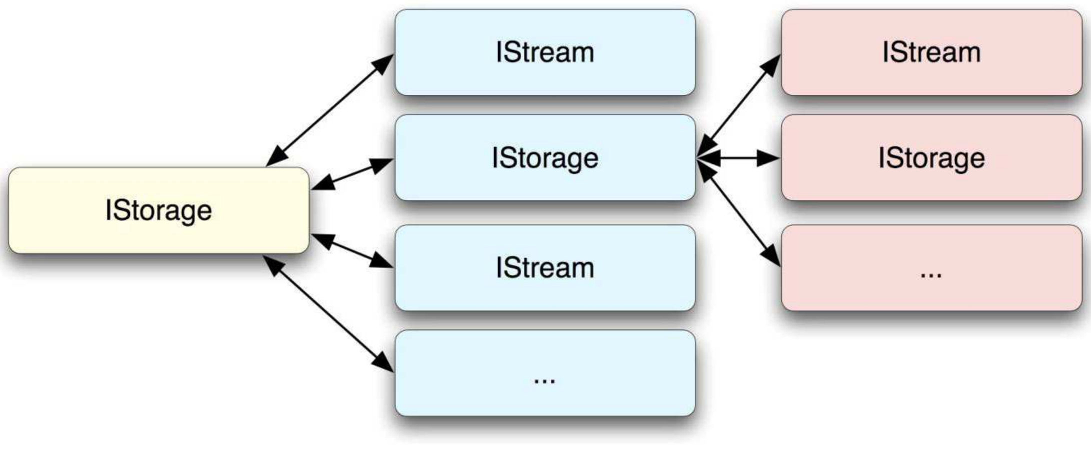


Figure 20: An example of a storage file's contents.

```
MIDL_INTERFACE("0000010c-0000-0000-C000-000000000046")
Persist : public IUnknown
{
public:
        virtual HRESULT STDMETHODCALLTYPE GetClassID(
                /* [out] */ __RPC__out CLSID *pClassID) = 0;
};
```

- `IPropertyBag`
- `IPropertyBag2`

COM objects support serialization by implementing one of several well-known persistence interfaces. Each of these persistence interfaces are specializations of the `IPersist` interface, which has the following definition:

Each subclass of `IPersist` has methods named `Load()` and `Save()`, which serialize the data and resurrect the data, respectively. The differentiator between these subclasses is the type of interface that holds the persisted data. Table **??** lists the persistence interfaces, and the argument type that each respective interface uses to hold the data. Figure 21 visually depicts the inheritance hierarchy of these interfaces.

When a host program wishes to serialize an object, it will query that object for a persistence interface. If successful, the application will then call the `Save()` method, passing a pointer to one of the previously-discussed storage interfaces (`IStream`, `IStorage`, `IFile`, etc). Later, when a host program wishes to resurrect the object from its persistent state, it

Table 3: Persistence Interfaces correlated to Data Interfaces

Persistence Interface	Argument that Holds the Data
`IPersistFile`	An `LPCOLESTR` that designates a standard file path
`IPersistMemory`	An `LPVOID` that is a fixed-size memory buffer
`IPersistMoniker`	An `IMoniker` interface
`IPersistPropertyBag`	An `IPropertyBag` interface
`IPersistPropertyBag2`	An `IPropertyBag2` interface
`IPersistStorage`	An `IStorage` interface
`IPersistStream`	An `IStream` interface
`IPersistStreamInit`	An `LPSTREAM` interface

will once again retrieve the object's persistence interface, and call the `Load()` method. The object resurrected from the persistence data should be equivalent to the object that was previously saved.

3.3.1.9 Implementing COM Persistence in the ATL

Developers of COM objects are free to implement their own persistence interfaces. If these developers choose to write their own code for the interface, they would manipulate the interface that stores the persistence data by reading and writing data in an arbitrary format. However, most developers choose to use template classes provided in the Microsoft ATL, when there is template code to do so, avoiding the extra work it would require to implement these interfaces. Version nine of the Microsoft ATL has template classes for the following persistence interfaces:

- `IPersist`
- `IPersistPropertyBag`
- `IPersistStorage`
- `IPersistStreamInit`

The template code requires a programmer to define a series of properties, known as a property map, which the persistence interface will use as a boiler plate for serializing and resurrecting the object in question. This property map is a terminated array of structures that list the properties for the control that must be serialized and resurrected, and should be

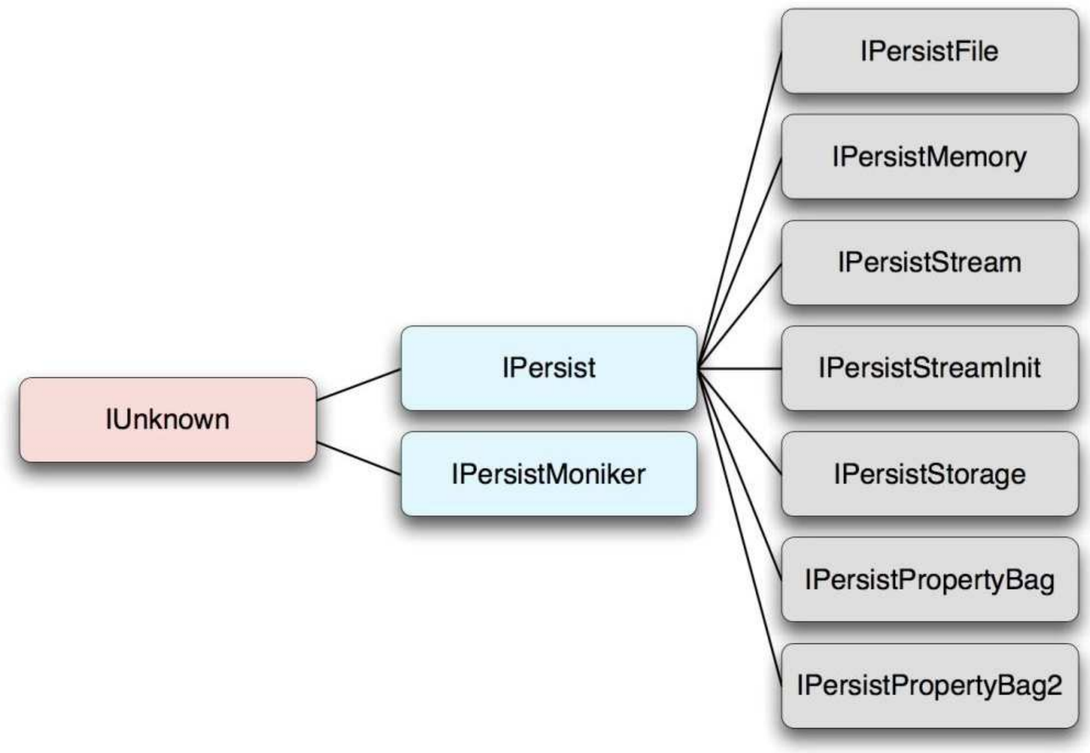


Figure 21: The inheritance hierarchy of persistence interfaces.

made explicit enough to guarantee that the object, once serialized, will be equivalent to an object that is resurrected from the data. Version nine of the ATL includes various macros to aid a programmer when defining these properties and include macros from the following list.

- `BEGIN_PROPERTY_MAP`
- `BEGIN_PROP_MAP`
- `PROP_ENTRY`
- `PROP_ENTRY_EX`
- `PROP_ENTRY_TYPE`
- `PROP_ENTRY_TYPE_EX`
- `PROP_PAGE`

```
struct ATL_PROPMAP_ENTRY {
        LPCOLESTR szDesc;
        DISPID dispid;
        const CLSID* pclsidPropPage;
        const IID* piidDispatch;
        DWORD dwOffsetData;
        DWORD dwSizeData;
        VARTYPE vt;
};
```

Figure 22: Type definition of ATL_PROPMAP_ENTRY

- PROP_DATA_ENTRY
- END_PROPERTY_MAP
- END_PROP_MAP

Each of the previously mentioned macro functions take various arguments and use them to define an ATL_PROPMAP_ENTRY structure. The following code is the structure definition taken from version nine of the ATL.

The elements in the ATL_PROPMAP_ENTRY structure are all quite important to understand, and are summarized in Table ??.

The macro functions for defining properties use arguments supplied to the function to set certain ATL_PROPMAP_ENTRY elements, and will set others to a default state. Depending on the elements that have non-default values, the template code responsible for the persistence operations will use slightly differing strategies when serializing and resurrecting the data. Both BEGIN_PROPERTY_MAP and BEGIN_PROP_MAP will include code that starts to define the structure; however, the former will automatically include X and Y position information within the property map. END_PROP_MAP and END_PROPERTY_MAP are macro functions that will include a terminating ATL_PROPMAP_ENTRY element and end the structure definition. Between BEGIN_PROPERTY_MAP or BEGIN_PROP_MAP, and END_PROP_MAP or END_PROPERTY_MAP, are ATL_PROPMAP_ENTRY instances that describe the properties of a COM object. PROP_ENTRY and PROP_ENTRY_EX both define a property using the property's name, display id, and a property page that can be used to set the property. PROP_ENTRY_TYPE and PROP_ENTRY_TYPE_EX define the same information as PROP_ENTRY and PROP_ENTRY_EX;

Table 4: A listing of the elements of the ATL_PROPMAP_ENTRY structure and the purpose they serve.

Element Name	Element Purpose
`szDesc`	Unicode string that uniquely identifies the property name
`dispid`	32-bit integer that uniquely identifies the property within the object
`pclsidPropPage`	Pointer to a COM class id that identifies a COM class that offers a GUI interface to set and retrieve the property within the control
`piidDispatch`	Pointer to a COM interface id that describes an interface that inherits from `IDispatch`, which can be used to set the property through the Invoke method of the interface
`dwOffsetData`	32-bit value that specifies the property's memory offset from the beginning of the object
`dwSizeData`	32-bit value that specifies the number of bytes that have been allocated in the object to hold the property's data
`vt`	16-bit value that specifies the property's type

however they also require an explicit variant type that is expected when dealing with the property. The "`_EX`" suffix designates that the macro function also expects an explicit dispatch interface id that should be used when setting or getting the property's value. The `PROP_DATA_ENTRY` macro requires a unique string identifier for the property, the name of the class's member that will be used to store the property, and the type of variant that's expected for the property. Internally, the `PROP_DATA_ENTRY` macro uses the offsetof and sizeof structure to explicitly define `dwOffsetData` and `dwSizeData` within the `ATL_PROPMAP_ENTRY` structure. `PROP_PAGE` is used to specify a COM class id that offers a GUI interface, which can manipulate the properties of an object. To help illustrate the use of property maps in C code and how properties are read from a persisted state, we'll briefly present an example COM object called HelloCom. HelloCom is a simple ActiveX control that can store a person's first and last names. The properties will have the names `NameFirst` and `NameLast`. The following C++ code snippet shows portions of code for the HelloCom control that are relevant for implementing persistence.

```
class HelloCom :
        public IPersistStreamInitImpl<HelloCom>,
        public IPersistStorageImpl<HelloCom>,
        public IPersistPropertyBagImpl<HelloCom>
{
public:
BEGIN_PROP_MAP(HelloCom)
        PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
        PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
        PROP_ENTRY("NameFirst", 1, CLSID_HelloComCtrl)
        PROP_ENTRY_TYPE("NameLast", 2, CLSID_HelloComCtrl, VT_BSTR)
END_PROP_MAP()
};
```

Figure 23: Class definition for `HelloCom`

If the application is loading the persistence data from a binary stream, then the application would query for the `IPersistStreamInit` interface and would receive a vtable pointing to the `IPersistStreamInitImpl` template class. Next, the application will call the `Load()` method, passing in an `IStream` object that will be used to read the persistence data. Prior to any of the serialized data in a stream, a version number is stored in order to deal with backwards compatibility issues. So, the first four bytes in the stream will be a little-endian representation of the ATL version that was used to compile the control. In Visual Studio 2008, this value is 0x00000900. As long as the value is less-than or equal-to the version of the ATL used to compile the control, processing can resume, otherwise, an error is signaled. After the versioning information has been processed, the properties themselves can then be retrieved from the stream. The bytes immediately after the version number in the stream in this case would be two 4-byte little-endian representations of the `_cx` and `_cy` elements. Since these elements were declared with the `PROP_DATA_ENTRY` macro, these 32-bit values will be written directly to the memory offset in the class where the `m_sizeExtent.cx` and `m_sizeExtent.cy` values reside. Following these values, we will encounter the serialized representation of `NameFirst`. Since `NameFirst` is declared in the property map using the `PROP_ENTRY()` macro, which contains no data type, the type information needs to be retrieved from the stream. Therefore, the first two bytes in the stream would be an unsigned 16-bit value of 0x0008, representing the variant type `VT_BSTR`. Next would come an unsigned 32-bit value specifying the length of the string. If the name were "Example", then

Offset	Hexadecimal representation of bytes	Description
0x00	00 09 00 00	Version nine of the ATL
0x04	00 01 00 00	The `_cx` value is 256
0x08	00 01 00 00	The `_cy` value is 256
0x0C	08 00	NameFirst is stored as a `VT_BSTR`
0x0E	0C 00 00 00	`NameFirst` is 12 characters long
0x12	46 00 69 00 72 00 73 00 74 00 00 00	`NameFirst` is equivalent to "First"
0x1E	0A 00 00 00	`NameLast` is 10 bytes long
0x22	4C 00 61 00 73 00 74 00 00 00	`NameLast` is equivalent to "Last"

Figure 24: A listing of the elements contained in a stream for the fictitious HelloCom example

the value of this 32-bit integer specifying the size would equal 0x10; seven 2-byte characters plus a terminating null. The next values would be the characters that represent the name, followed by a terminating 16-bit value of 0x0000. `NameLast` would come next, and would be specified identically to `NameFirst`, except that the 16-bit variant type specifier would be absent in the stream, since the type is explicitly declared in the property map using the `PROP_ENTRY_TYPE()` macro. Table 24 shows an example of the stream described in the previous paragraphs, with hexadecimal values representing the value in the stream, an offset showing the position of the value in the stream, and a description of how the values should be interpreted.

3.3.1.10 COM Persistence in Microsoft Internet Explorer

Microsoft Internet Explorer uses persistence when assigning values to properties of ActiveX objects. The six main interfaces used by Internet Explorer, ordered by preference, are `IPersistPropertyBag`, `IPersistMoniker`, `IPersistFile`, `IPersistStreamInit`, `IPersistStream`, and `IPersistStorage`. The browser will attempt to retrieve an interface pointer to each persistence interface in sequence until it is successful, or no interfaces have been found, at which point the operation fails. The first, and most familiar, persistence interface is `IPersistPropertyBag`. `IPersistPropertyBag` has been specifically designed to allow persistent objects to be embedded within HTML. Take, as an example, the following HTML code that embeds Microsoft Media Player within a web page.

```
<OBJECT id="VIDEO" CLASSID=
        "CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6" >
                <PARAM NAME="URL" VALUE="MyVideo.wmv">
                <PARAM NAME="enabled" VALUE="True">
                <PARAM NAME="AutoStart" VALUE="False">
                <PARAM name="PlayCount" value="3">
                <PARAM name="Volume" value="50">
                <PARAM NAME="balance" VALUE="0">
                <PARAM NAME="Rate" VALUE="1.0">
                <PARAM NAME="Mute" VALUE="False">
                <PARAM NAME="fullScreen" VALUE="False">
                <PARAM name="uiMode" value="full">
</OBJECT>
```

The `<PARAM>` tags that appear within the `<OBJECT>` tag represent the COM object's property names and persisted values. When Internet Explorer parses a web page and encounters these `<PARAM>` tags, it first creates a `PropertyBag` class and queries for the `IPropertyBag` interface. Next, it will parse the name and value parameters of the `<PARAM>` HTML tag and call the `Write()` method on the `IPropertyBag` interface, supplying the name and a string representation of the value for the property it has parsed. Once Internet Explorer has loaded all of the `<PARAM>` tags into a property bag, it will query the COM object (In the above example, a Media Player object) for an `IPersistPropertyBag` interface. Internet Explorer will then call the `Load()` method of the `IPersistPropertyBag` interface, passing the `PropertyBag` that was parsed from the HTML. The `Load()` method of the COM object will then convert the properties from a string representation into the object's preferred representation, and subsequently save the converted representation within the COM object. This strategy is employed by Internet Explorer to resurrect the object from a persistent state when it encounters the above HTML. The reciprocal operation to the resurrection operation, serialization, is most commonly encountered when using the innerHTML attribute of an object. Consider the following JavaScript code, used in the same web page as the above HTML.

```
<script language="JavaScript"> alert(VIDEO.innerHTML); </script>
```

Upon processing the previous JavaScript, the web page will alert the user with a message box with HTML formatted text similar to the following example:

```
<PARAM NAME="URL" VALUE="./MyVideo.wmv">
<PARAM NAME="rate" VALUE="1">
<PARAM NAME="balance" VALUE="0">
<PARAM NAME="currentPosition" VALUE="0">
<PARAM NAME="defaultFrame" VALUE="">
<PARAM NAME="playCount" VALUE="3">
<PARAM NAME="autoStart" VALUE="0">
<PARAM NAME="_cx" VALUE="6482">
<PARAM NAME="_cy" VALUE="6350">
```

When Internet Explorer serializes an object using a `PropertyBag`, it first creates an instance of the `PropertyBag` class. Next, it queries the object to be persisted for the `IPersistPropertyBag` interface. Once the interface is retrieved, Internet Explorer calls the `Save()` method, passing the `PropertyBag` class instance. Finally, Internet Explorer will serialize the `PropertyBag` class into a format that is compatible with HTML standards. The second, less common, way of inserting persistence data into a control over Internet Explorer is through the use of the data parameter of the `<OBJECT>` tag. An example of this type of persistence is shown in the following HTML.

```
<OBJECT
        id="VIDEO"
        CLASSID="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6"
        data="./persistence_data"
        type="application/x-oleobject"
/>
```

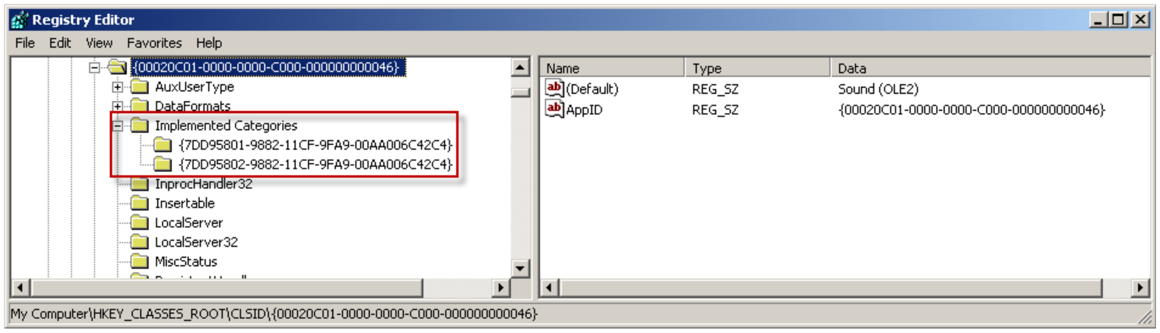
In the example above, instead of using `<PARAM>` tags, the persistence data is communicated through the data parameter of the object tag. When Internet Explorer encounters an object tag in this format, it follows a complex strategy to resurrect the object from the serialized data. Internet Explorer will first check the file name specified in the data parameter to see if the file name extension is equal to ".ica", ".stm", or ".ods". If the extension is one of these, then it creates an IStream that can read binary data from the supplied file URL. Internet Explorer will then create an instance of the object specified in the first sixteen bytes of the file, or, if those sixteen bytes are zero, the CLASSID parameter in the object tag and query for the `IPersistStream` interface. If the interface is successfully retrieved, Internet Explorer will then call the `Load()` method of the interface, passing in the `IStream`. Next, the COM object will parse the stream and convert the binary data into the preferred representation of each property. Once these operations are finished, Internet Explorer will have a fully resurrected COM object. If the filename does not match one of the well-known extensions, Internet Explorer does some extra work to determine what type of persistence interface to use for the COM object and corresponding persistence data. First, Internet Explorer will query the COM object for an `IPersistFile` interface. If the interface is successfully retrieved, it will call the `Load()` method of the COM object's interface, passing in a file path. It is then the COM object's responsibility to open the file and parse the data. If the object does not support the `IPersistFile` interface, Internet Explorer will use the URL in the data value, and create an `IStream` object. Next, it will query the COM object for an `IPersistStreamInit` interface. If this operation is successful, then Internet Explorer will call the `Load()` method of the `IPersistStreamInit` interface, passing in the `IStream` object. If the COM object doesn't support the `IPersistStreamInit` interface, it will then attempt to query the object for an IPersistStream interface. If the object implements this interface, then Internet Explorer will call the `Load()` method of the `IPersistStream` interface, passing in the `IStream` object. If these operations are successful, then the COM object's `IPersistStreamInit` or `IPersistStream` interface is charged with resurrecting the properties from the given persistence data. If the COM object does not implement either `IPersistStreamInit`, or `IPersistStream`, or the `Load()` method returns with a failure, then Internet Explorer will attempt to load the URL as a compound OLE document by calling `StgOpenStorage()` from OLE32. If `StgOpenStorage()` returns with a successful value, then Internet Explorer will query the COM object for an `IPersistStorage` interface. If the COM object indeed implements the `IPersistStorage` interface, then Internet Explorer will call the `Load()` method of the interface, passing in an `IStorage` object. From here it is, again, the responsibility of the COM object to parse the data contained in the `IStorage` object.

3.3.2 Microsoft ActiveX

ActiveX is a technology derived from Microsoft's COM technology. It is utilized to create plugins that can be exposed to runtime engines (such as JavaScript and VBScript) to provide additional capabilities to the host application. Understanding the types of vulnerabilities that will be explored later in this dissertation requires an in-depth understanding of some of the COM / Automation architecture. As such, we will present an overview of the relevant technologies in this section. We will also explore the concept of "persistent objects", which are serialized COM objects that can be optionally embedded within web pages. It will be shown in section three how persistent COM objects can be used to not only target vulnerabilities in various COM marshaling components, but also undermine browser security features in certain scenarios.

3.3.2.1 Plugin Registration

ActiveX controls are a specialization of COM objects, and as such have an entry within the system registry describing the relevant instantiation information. Like any other COM object, each ActiveX object is identified by a globally unique Class ID (CLSID), and is located in the registry at HKEY_CLASSES_ROOT\CLSID\{<CLSID>}. Objects can also be installed on a per-user basis, using the HKEY_CURRENT_USER portion of the registry. Since COM objects are used so pervasively throughout the Windows OS, Internet Explorer (IE) needs a way of restricting which COM objects are allowed to be launched through the web browser. The semantics of the safety mechanisms have gradually become more granular over time, and will briefly be described here.


Registry Editor

File Edit View Favorites Help

```
{00020C01-0000-0000-C000-000000000046}
    AuxUserType
    DataFormats
    Implemented Categories
        {7DD95801-9882-11CF-9FA9-00AA006C42C4}
        {7DD95802-9882-11CF-9FA9-00AA006C42C4}
    InprocHandler32
    Insertable
    LocalServer
    LocalServer32
    MiscStatus
```

Name	Type	Data
(Default)	REG_SZ	Sound (OLE2)
AppID	REG_SZ	{00020C01-0000-0000-C000-000000000046}

My Computer\HKEY_CLASSES_ROOT\CLSID\{00020C01-0000-0000-C000-000000000046}

Figure 25: ActiveX control marked as “Safe for Initialization” (SFI) and “Safe For Scripting” (SFS)

3.3.2.2 ActiveX Plugins: Safety Controls

IE has several mechanisms for determining whether an ActiveX object has permission to run. Safety permissions for controls are divided into two categories: initialization and scripting. Initialization safety refers to whether or not the control is allowed to be instantiated based on data from a persistent COM stream (discussed in depth shortly). Scripting safety refers to whether the control may be manipulated via scripting APIs exposed at runtime. Registry Controls The first and most well-known method to mark a control as Safe For Scripting (SFS) or Safe For Initialization (SFI) is to add specific subkeys below the entry for the control in the registry. Two values can be added under the “Implemented Categories” subkey to mark the control SFS and SFI respectively. These values are 7DD95801-9882-11CF-9FA9-00AA006C42C4 (CATID_SafeForScripting) and 7DD95802-9882-11CF-9FA9-00AA006C42C4 (CATID_SafeForInitialization) respectively. Figure 25 shows an example of a control using these categories.

Controls may programmatically register themselves for these categories using the `StdComponentCategoriesMgr` object. The `ICatRegister` interface contains a `RegisterClassImplCategories()` method, which can be used to manipulate the category registration information for any given COM object. Internally, the `StdComponentCategoriesMgr` updates the registry with the above information. Internet Explorer utilizes the `StdComponentCategoriesMgr` object as well, but for enumeration rather than registration. The ICatInformation interface provides a function named `IsClassOfCategories()`, which IE can call to determine if a control is SFS or SFI. Again, this operation internally queries the above mentioned registry location to determine which controls the object implements.

3.3.2.3 `IObjectSafety` *Control*

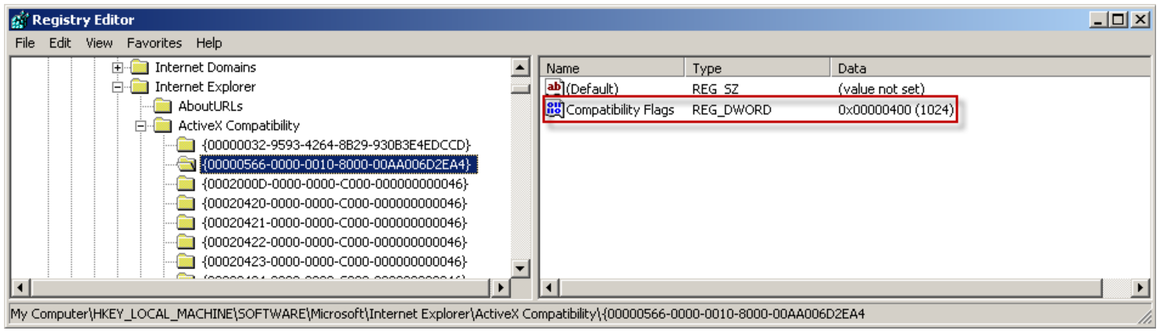
An alternative method exists to mark a control as SFS or SFI. An ActiveX control can provide support for either of these safety restrictions by implementing the `IObjectSafety` interface. In this scenario, the security capabilities for the control can be obtained by calling the `IObjectSafety::GetInterfaceSafetyOptions()` method, which has the following prototype:

```
HRESULT IObjectSafety::GetInterfaceSafetyOptions(
         REFIID riid, DWORD *pdwSupportedOptions,
         DWORD *pdwEnabledOptions );
```

This function will be called by IE to determine the supported set of safety options. If the interface appears to support the security options, IE will then call the `SetInterfaceSafetyOptions()` method of the `IObjectSafety` interface with the options that it would like the object to enforce. `SetInterfaceSafetyOptions` has the following prototype:

```
HRESULT IObjectSafety::SetInterfaceSafetyOptions(
         REFIID riid, DWORD dwOptionSetMask,
         DWORD dwEnabledOptions );
```

If `SetInterfaceSafetyOptions()` returns successfully, then the application can use the COM object knowing that the object intends to use the security options requested. The added value of this API over COM categories is that a control can offer more granular control over how it is used, since it is able to specify different security settings for different interfaces, based on which interface id was specified in the riid parameter for the method calls. Also, the `IObjectSafety` interface can execute native code to determine if the application that is creating the object can do so safely. A specific example of this type of functionality is the SiteLock template code provided by Microsoft. This template code allows the programmer to restrict ActiveX controls to a pre-determined list of URLs.



```
Internet Domains
Internet Explorer
    AboutURLs
    ActiveX Compatibility
        {00000032-9593-4264-8B29-930B3E4EDCCD}
        {00000566-0000-0010-8000-00AA006D2EA4}
        {0002000D-0000-0000-C000-000000000046}
        {00020420-0000-0000-C000-000000000046}
        {00020421-0000-0000-C000-000000000046}
        {00020422-0000-0000-C000-000000000046}
        {00020423-0000-0000-C000-000000000046}
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
Compatibility Flags	REG_DWORD	0x00000400 (1024)

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\ActiveX Compatibility\{00000566-0000-0010-8000-00AA006D2EA4
```

Figure 26: ActiveX Killbits in IE

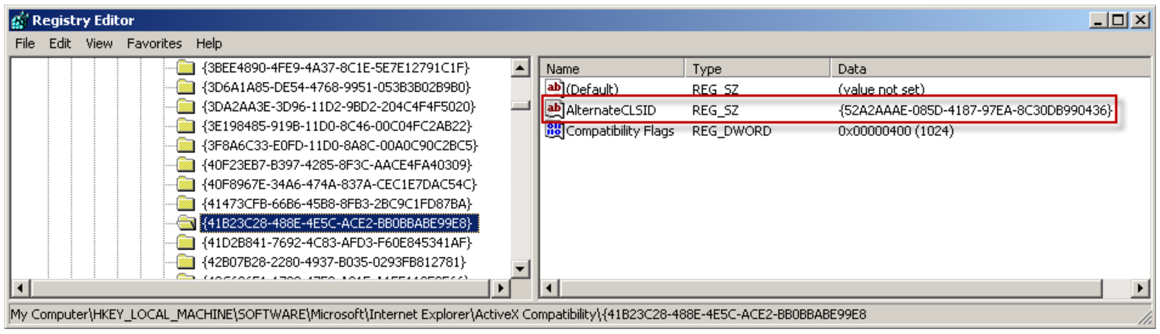
3.3.2.4 ActiveX Killbits

IE also implements an override to the standard safety features, allowing administrators to specifically ban the instantiation of selected controls within the browser. This is achieved by adding a subkey into the HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\ActiveX Compatibility registry location. The subkey added must have the CLSID of the control in question, and contain the DWORD value "Compatibility Flags", which has the "killbit" set (value 0x400). Figure 26 shows an example of a control with the killbit set.

When an application wishes to determine if the killbit is set, it will call the `CompatFlagsFromClsid()` function, which is exported from urlmon.dll. `CompatFlagsFromClsid()` has the following prototype:

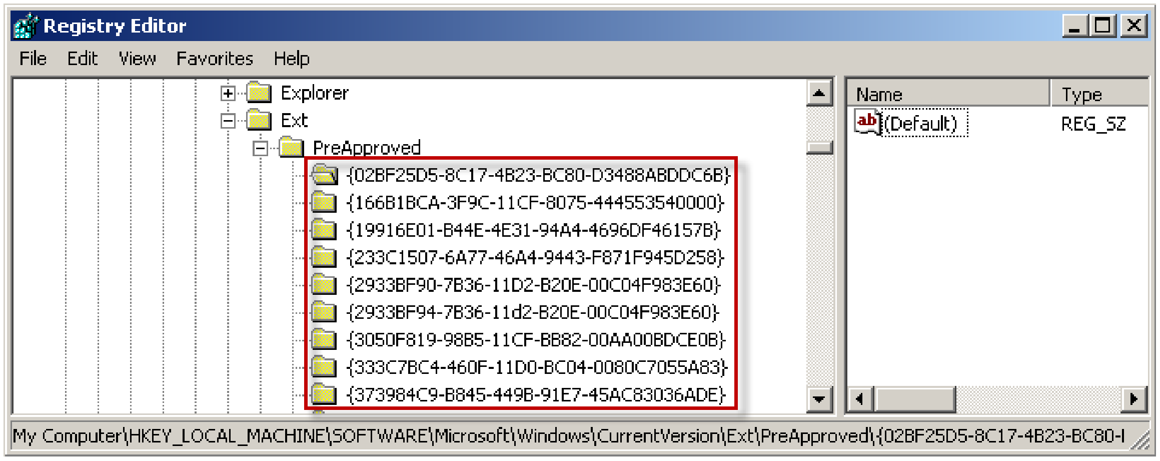
```
HRESULT CompatFlagsFromClsid( CLSID *pclsid,
        LPDWORD pdwCompatFlags, LPDWORD pdwMiscStatusFlags );
```

When the application calls this function, it will pass in the CLSID of the COM object it is interested in, and two `DWORD` pointers whose value will be equal to the compatibility and miscellaneous OLE flags for the object upon the successful return of the function. The application will then test to see if the 0x400 bit is set to determine if the control has the killbit set. If the killbit is set, then an entry may appear in the registry for an alternate class id. This alternate class id will be used in lieu of the original class id within Internet Explorer. Figure 27 shows a registry entry for a class id that uses an alternate class id. When dealing with the control in Figure 27, Internet Explorer will transparently translate requests for COM objects with a class id of {41B23C28-488E-4E5C-ACE2-BB0BBABE99E8} to the



```
{3BEE4890-4FE9-4A37-8C1E-5E7E12791C1F}
{3D6A1A85-DE54-4768-9951-053B3B02B9B0}
{3DA2AA3E-3D96-11D2-9BD2-204C4F4F5020}
{3E198485-919B-11D0-8C46-00C04FC2AB22}
{3F8A6C33-E0FD-11D0-8A8C-00A0C90C2BC5}
{40F23EB7-B397-4285-8F3C-AACE4FA40309}
{40F8967E-34A6-474A-837A-CEC1E7DAC54C}
{41473CFB-66B6-45B8-8FB3-2BC9C1FD87BA}
{41B23C28-488E-4E5C-ACE2-BB0BBABE99E8}
{41D2B841-7692-4C83-AFD3-F60E845341AF}
{42B07B28-2280-4937-B035-0293FB812781}

Name                 Type        Data
(Default)            REG_SZ      (value not set)
AlternateCLSID       REG_SZ      {52A2AAAE-085D-4187-97EA-8C30DB990436}
Compatibility Flags  REG_DWORD   0x00000400 (1024)

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\ActiveX Compatibility\{41B23C28-488E-4E5C-ACE2-BB0BBABE99E8
```

Figure 27: COM object with an alternate CLSID



```
Explorer
Ext
  PreApproved
    {02BF25D5-8C17-4B23-BC80-D3488ABDDC6B}
    {166B1BCA-3F9C-11CF-8075-444553540000}
    {19916E01-B44E-4E31-94A4-4696DF46157B}
    {233C1507-6A77-46A4-9443-F871F945D258}
    {2933BF90-7B36-11D2-B20E-00C04F983E60}
    {2933BF94-7B36-11d2-B20E-00C04F983E60}
    {3050F819-98B5-11CF-BB82-00AA00BDCE0B}
    {333C7BC4-460F-11D0-BC04-0080C7055A83}
    {373984C9-B845-449B-91E7-45AC83036ADE}

Name       Type
(Default)  REG_SZ

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Ext\PreApproved\{02BF25D5-8C17-4B23-BC80-I
```

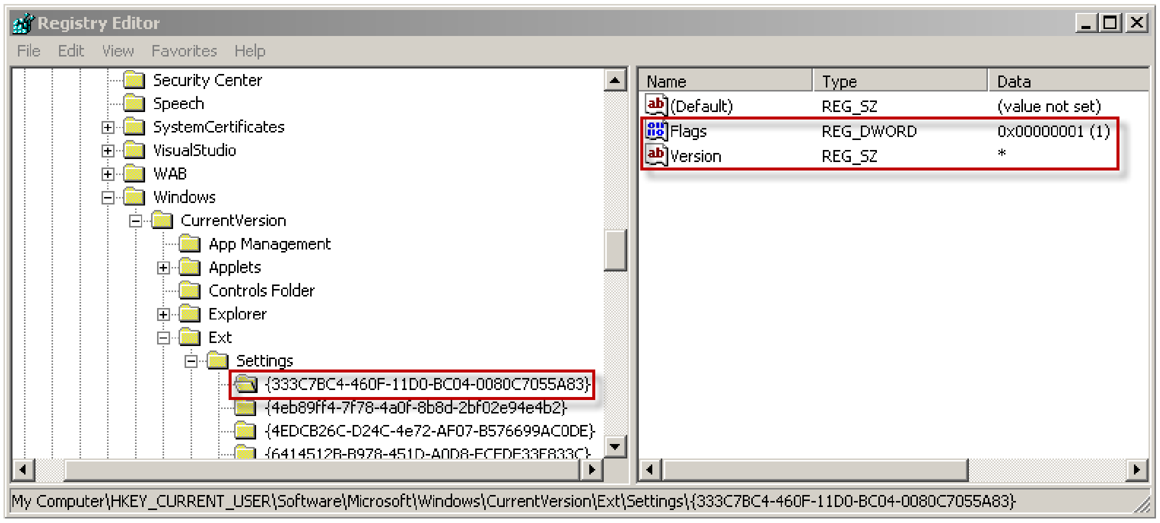
Figure 28: Excerpt of the preapproved list

class id of {52A2AAAE-085D-4187-97EA-8C30DB990436}.

3.3.2.5 *Preapproved List / ActiveX Opt-In*

Microsoft introduced a feature called ActiveX Opt-In with Internet Explorer 7. ActiveX Opt-In is designed to reduce the attack surface of the browser by prompting the user before a web page is allowed to instantiate an object that hasn't been loaded before in Internet Explorer, or wasn't installed by the user through Internet Explorer. Figure 28 shows the relevant area of the registry: HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Ext\PreApproved.

In a base installation of Windows there are a number of controls already on the preapproved list. However, there are far more controls that are safe for scripting or initialization that do not appear on this list. This functionality makes it more desirable to find flaws in controls on this list, rather than flaws in other controls.



```
Security Center
Speech
SystemCertificates
VisualStudio
WAB
Windows
  CurrentVersion
    App Management
    Applets
    Controls Folder
    Explorer
    Ext
      Settings
        {333C7BC4-460F-11D0-BC04-0080C7055A83}
        {4eb89ff4-7f78-4a0f-8b8d-2bf02e94e4b2}
        {4EDCB26C-D24C-4e72-AF07-B576699AC0DE}
        {6414512B-B978-451D-A0D8-FCFDF33E833C}
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
Flags	REG_DWORD	0x00000001 (1)
Version	REG_SZ	*

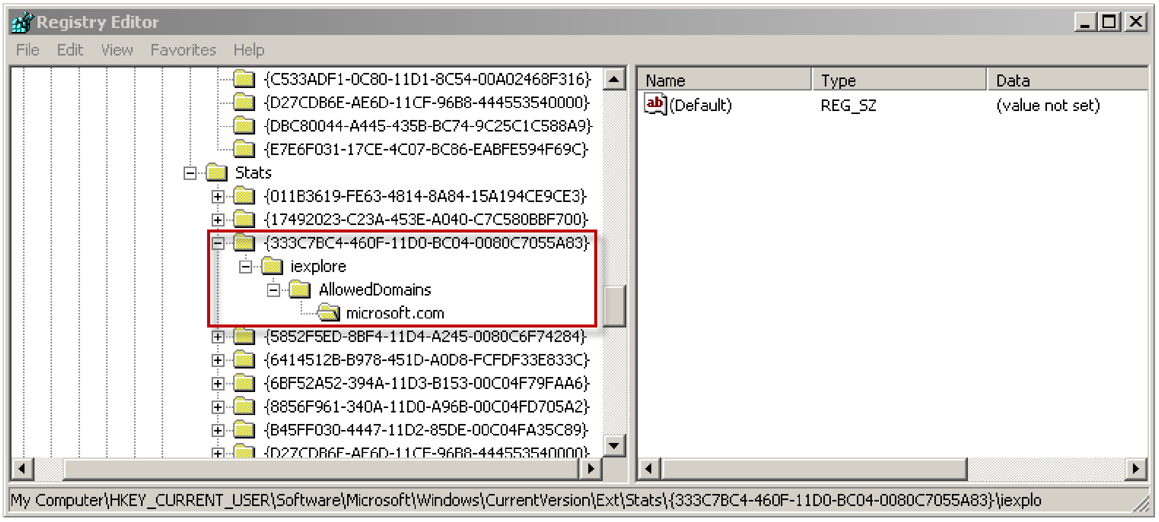
My Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Settings\{333C7BC4-460F-11D0-BC04-0080C7055A83}

Figure 29: Example of a control restricted from a single user

3.3.2.6 Per-User ActiveX Security

IE8 introduced a series of additional security capabilities related to secure browsing, including some refinements to ActiveX. Before these capabilities were added, control permissions that could be configured were configured on a per-machine basis. The new capabilities extend the per-machine killbit to a per-user level of granularity, and expand upon ActiveX opt-in by allowing Opt-In functionality based on the user and the domain. Traditionally, killbits have been used to effectively ban the instantiation of a control system-wide. This model is problematic in scenarios where a single user on a system of many users required the use of a particular control, but no others required it. Microsoft expanded upon killbits by introducing the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Settings\{CLSID}, where CLSID is the class id of the ActiveX control to restrict. By setting the Flags value of this key to "1", a control will be restricted for a single user. Figure 29 shows the Tabular Data Control disabled in this area of the registry.

Restricting ActiveX controls to certain domains allows the user to have more granular control over ActiveX security. Originally, SiteLock was the only method that allowed domain restriction, which was not configurable by the end user. This new per-domain restriction is managed in the registry by adding keys for specific allowed domains



```
{C533ADF1-0C80-11D1-8C54-00A02468F316}
{D27CDB6E-AE6D-11CF-96B8-444553540000}
{DBC80044-A445-435B-BC74-9C25C1C588A9}
{E7E6F031-17CE-4C07-BC86-EABFE594F69C}
Stats
  {011B3619-FE63-4814-8A84-15A194CE9CE3}
  {17492023-C23A-453E-A040-C7C580BBF700}
  {333C7BC4-460F-11D0-BC04-0080C7055A83}
    iexplore
      AllowedDomains
        microsoft.com
  {5852F5ED-8BF4-11D4-A245-0080C6F74284}
  {6414512B-B978-451D-A0D8-FCFDF33E833C}
  {6BF52A52-394A-11D3-B153-00C04F79FAA6}
  {8856F961-340A-11D0-A96B-00C04FD705A2}
  {B45FF030-4447-11D2-85DE-00C04FA35C89}
  {D27CDB6E-AE6D-11CF-96B8-444553540000}
```

Name	Type	Data
(Default)	REG_SZ	(value not set)

My Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Stats\{333C7BC4-460F-11D0-BC04-0080C7055A83}\iexplo

Figure 30: Example of a control approved to be run from the Microsoft.com domain

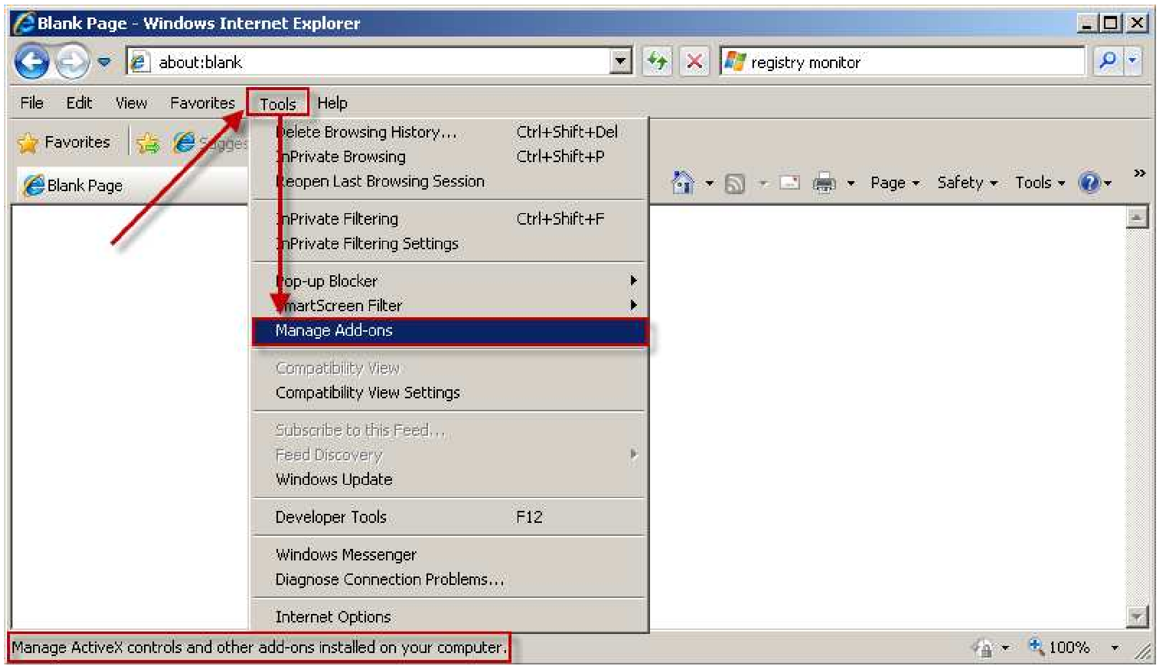
to HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Stats\{CLSID}\iexplore\AllowedDomains. A key for all domains can be added here by using the name "*", as opposed to a specific domain. Per-Domain opt-in controls reduce the attack surface by requiring the user to approve the use of an ActiveX control before it is ran in the context of an unfamiliar domain. In effect, this would require an attacker to insert malicious web content onto a trusted domain in order to surreptitiously exploit the ActiveX control. Figure 30 shows the Tabular Data Control configured to run within the microsoft.com domain without prompting.

3.3.2.7 Internet Explorer Permission GUI

In addition to providing restriction capabilities, Microsoft enhanced the Internet Explorer UI by adding an interface that allows the user to easily configure ActiveX control permissions without having to modify the registry. Figure 31 shows how to access the Add-on Manager interface, and Figure 32 shows how to find DLLs that are allowed to run in the browser without permission.

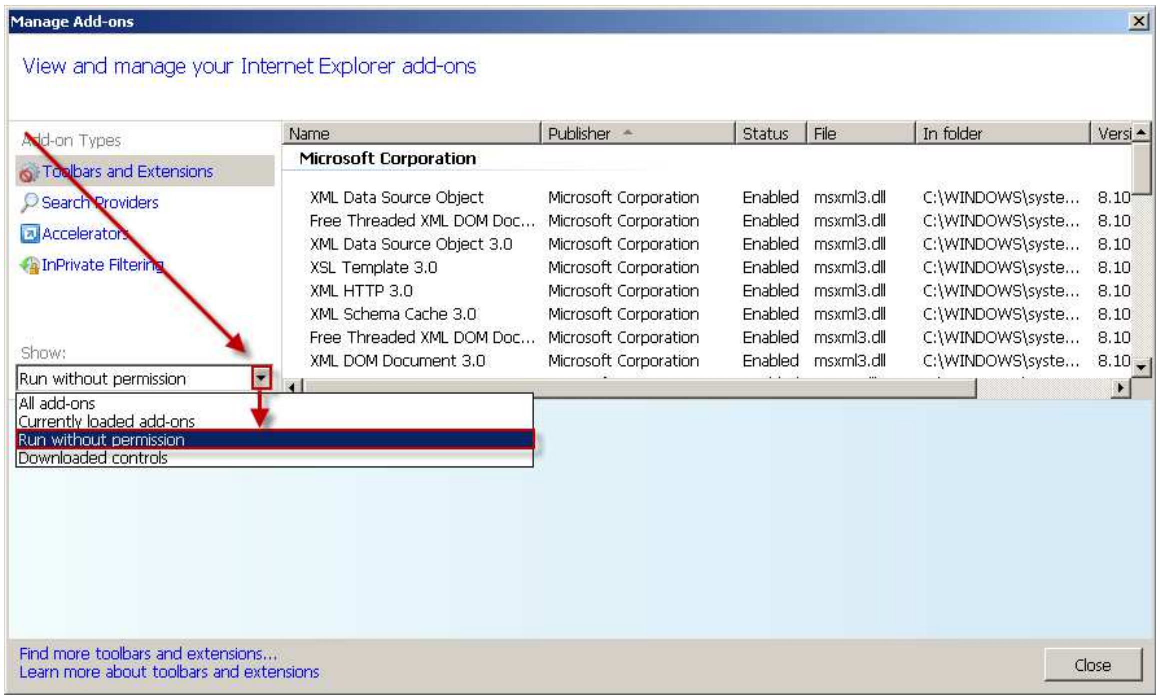
3.3.2.8 ActiveX Safety Wrap-Up

ActiveX has many methods to restrict which controls may load, and how they can be acted upon under a given context. One reason may very well be that, as interoperability has


Blank Page - Windows Internet Explorer
about:blank
registry monitor
File
Edit
View
Favorites
Tools
Help
Favorites
Blank Page
Delete Browsing History... Ctrl+Shift+Del
InPrivate Browsing Ctrl+Shift+P
Reopen Last Browsing Session
InPrivate Filtering Ctrl+Shift+F
InPrivate Filtering Settings
Pop-up Blocker
SmartScreen Filter
Manage Add-ons
Compatibility View
Compatibility View Settings
Subscribe to this Feed...
Feed Discovery
Windows Update
Developer Tools F12
Windows Messenger
Diagnose Connection Problems...
Internet Options
Page
Safety
Tools
100%

Manage ActiveX controls and other add-ons installed on your computer.

Figure 31: Navigating to the Add-on Manager



Name	Publisher	Status	File	In folder	Versi
Microsoft Corporation					
XML Data Source Object	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
Free Threaded XML DOM Doc...	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
XML Data Source Object 3.0	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
XSL Template 3.0	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
XML HTTP 3.0	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
XML Schema Cache 3.0	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
Free Threaded XML DOM Doc...	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10
XML DOM Document 3.0	Microsoft Corporation	Enabled	msxml3.dll	C:\WINDOWS\syste...	8.10

Figure 32: Operations to display controls that will run without permission

increased in applications, so too has opportunities for attackers. Under this premise, ActiveX security has evolved in an attack-response fashion and has led to a somewhat fractured security architecture. In later sections, we will show an attack that allows some of these restrictions to be bypassed, mostly as a result of Microsoft adding security features to the browser in an ad-hoc fashion rather than having established a robust security architecture from the outset.

Chapter IV

INSTANTIATION PROBLEMS

As discussed in Chapter 1, problems with derived types can occur at all stages of their lifecycle. This chapter discusses the issues associated with the instantiation of objects.

4.1 C++ Constructor Failure

One of the shortcomings of C++ is there is no way to return an error code from a constructor. In fact, using a return statement anywhere in a constructor will cause at a minimum a warning during compilation, and in the worse case an error. In any case, the return statement is useless since the only thing that can be returned from a constructor is an instantiated object. Under certain conditions, constructors have to perform some pretty complex operations. Consider the code in Figure 33, for example. Here we can see the constructor makes a call to a web service to retrieve a value for `some_property`. If that web service request fails, the value of `some_property` will be invalid.

There are cases where the developer instantiating the object may not know what a valid value for the property might be. For example, in the code in Figure 33, if the web service were to return an integer, and any integer value was valid, there would be no clear way to determine whether the value was valid. The correct way to handle this situation would be to catch the exception thrown by `curlpp`, or to pass the exception to the code that is instantiating the object. The code in Figure 34 shows how this could be done.

4.1.1 Detecting C++ Constructor Failure With Static Analysis

Detecting constructor failure is an area for future research. This is a very difficult problem as there are many ways that a constructor can fail. The example in the previous subsection is only one such type of failure.

```
#include <curlpp/cURLpp.hpp>
#include <curlpp/Options.hpp>

class Class1 {
        std::string some_property;

        Class1();
        ~Class1() { };
};

Class1::Class1()
{
        some_property << curlpp::options::Url(
                std::string(
                "http://my-rest-interface.com/some_property"));
}
```

Figure 33: Example C++ code with a failure in the constructor

```
#include <curlpp/cURLpp.hpp>
#include <curlpp/Options.hpp>

class Class1 {
        std::string some_property;

        Class1();
        ~Class1() { };
};

Class1::Class1()
{
        try
        {
                some_property << curlpp::options::Url(
                        std::string(
                        "http://my-rest-interface.com/some_property"));
        }

        catch(curlpp::RuntimeError & e)
        {
                std::cout << e.what() << std::endl;
        }
}
```

Figure 34: Example C++ code with a failure in the constructor with proper exception handling

4.2 COM Retention Failure

The Microsoft plugin architecture makes extensive use of COM objects and `VARIANT`s to define and pass objects between the various components within the browser. Indeed, JavaScript objects are represented natively in the language runtime as COM objects, whereas VBScript objects are represented as `VARIANT`s. A method or property exposed by an ActiveX object is accessed by calling the `IDispatch::Invoke()` method of the object, which receives parameters to the destination function as an array of `VARIANT`s. (Note that with ActiveX controls, properties are actually exposed as a pair of method calls that have names of the form `get_XXX()` and `put_XXX()`, where XXX is the name of the property. These two functions retrieve and set the property respectively.) Objects contained within the `VARIANT`s can really be any type and value, but most commonly they are either primitive types (such as integers or strings), or COM interfaces that represent complex objects. Since JavaScript represents objects internally as `IDispatch` (or more accurately, `IDispatchEx`) COM interfaces, `VT_DISPATCH VARIANT`s will be the most common COM-based `VARIANT`s passed to typical controls, in the context of a browser. COM objects maintain an internal reference count, and it is manipulated externally by the `IUnknown::AddRef()` and `IUnknown::Release()` methods, which increment and decrement the reference count respectively. Once the reference count reaches 0, the object will delete itself from memory. Object retention errors in ActiveX controls are a result of mismanagement of the object's reference count. This section describes the typical mistakes made by developers when dealing with objects whose lifespan is greater than the scope of a single function call.

4.2.1 ActiveX Object Retention Attacks I: No Retention

The most obvious mistake a control can make with regard to object retention is to neglect to add to the reference count of a COM object that it intends to retain. When an ActiveX function takes a COM object as a parameter, the marshaling layer has already called `IUnknown::AddRef()` on the received object to ensure that it will not be deleted by competing threads. However, the marshaller will also release the interface after the plugin function has returned. Therefore, a plugin object wishing to retain an instance of a COM

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
        m_pCallback = pCallback;
        return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
        if(out == NULL || *out == NULL || m_pCallback == NULL)
                return E_INVALIDARG;

        *out = m_pCallback;

        return S_OK;
}
```

Figure 35: Code demonstrating the failure to call `AddRef()`

object beyond the scope of a method must call the `IUnknown::AddRef()` function before the method returns. Calling `IUnknown::QueryInterface()` is also sufficient, as this function will (or at least, should) call `IUnknown::AddRef()` for the object as well. Failure to call either of these functions can result in potential stale pointer vulnerabilities. The code in Figure 35 shows an example of such a problem. The `put_MyProperty()` function in this code stores an `IDispatch` pointer which can later be retrieved by the client application using the `get_MyProperty()` function. However, since `AddRef()` is never used, there is no guarantee that the `pCallback` function will still exist when the property is read back by the client. If every other reference to the object is removed, the object will be de-allocated, leaving `m_pCallback` pointing to stale memory.

4.2.2 Detecting COM Retention Failures With Static Analysis

Detecting COM retention failures is an opportunity for future research. There is a direct correlation to the Automatic Reference Counting technology employed by Apple's Xcode to prevent retention failures of objective-C objects[16]. While this solution is not perfect, it has greatly reduced the number of app stability issues introduced by dangling object pointers.

4.3 COM Initialization Failure

Despite being a relatively simple data structure to manipulate, `VARIANT`s lend themselves to misuse in certain scenarios due to the deceptive nature of parts of the API. One of the key mistakes the authors uncovered when researching `VARIANT` usage for this work is the mismatching of `VarintInit()` and `VariantClear()` calls. As we mentioned earlier in the paper, the `VariantInit()` function is used to initialize a `VARIANT` structure by setting the vt member to `VT_EMPTY`. Conversely, `VariantClear()` will free the data associated with a `VARIANT`, taking into account what type of data is being stored there. It will subsequently set the type value of the `VARIANT`s to `VT_EMPTY`. The important thing to notice here is that any code path that exists where `VariantClear()` is called on a `VARIANT` that has not been initialized correctly can lead to potential security problems. This is because `VariantClear()` will read the uninitialized vt member of the `VARIANT` and use that to decide how to operate on the uninitialized `VARIANT` value. For example, if the vt member was `VT_DISPATCH` (0x0009), `VariantClear()` would take the data member from the `VARIANT` and dereference it to make an indirect call, since the process of deleting an `IDispatch` object involves calling the `IDispatch::Release()` function. Omission of the `VariantInit()` function creates a condition not unlike the memory management analog of freeing a block of memory without first allocating it, with two key differences: 1. Double `VariantClear()` is not the same as double `free()` – since `VariantClear()` sets the `VARIANT` type to `VT_EMPTY`, any subsequent calls to `VariantClear()` for the same `VARIANT` will have no effect, and 2. Omission of `VariantInit()` is more likely than `free()` without `malloc()`, because the code will still seemingly work correctly most of the time, even if the vulnerable code is exercised. This class of mistakes is really an uninitialized variable problem, with the additional caveat that the attacker needs to prime the appropriate memory area with useful data rather than specifying it directly. That is, the exploitability of these problems is very dependent on the residual data contained in the memory where the `VARIANT` was allocated. Under some conditions, this data is under the control of the attacker, while in other cases, the attacker simply needs to get lucky. An example `VariantInit()` omission vulnerability is shown in Figure 36.

As can be seen, a `VARIANT` located on the stack is manually initialized with the type `VT_DISPATCH`, and is presumably filled out with a pointer to an `IDispatch` interface after data has been successfully read from the source stream. However, if the `IStream::Read()` operation fails, the `VARIANT` is cleared, resulting in manipulating uninitialized stack data as

```
HRESULT MyFunc(IStream* pStream)
{
        VARIANT var;
        IDispatch* pDisp;
        HRESULT hr;

        var.vt = VT_DISPATCH;
        hr = pStream->Read(pDisp, sizeof(IDispatch *), NULL);

        if(FAILED(hr)) {
                VariantClear(&var);
                return hr;
        }

        . . .

        return hr;
}
```

Figure 36: Code sample demonstrating the omission of `VariantInit()`

if it pointed to an `IDispatch` interface. Although this seems like a relatively unlikely mistake to make, there are sometimes variations of the vulnerable code path that are slightly more subtle. One such example occurs when copying data between `VARIANT`s using the `VariantCopy()` function. The `VariantCopy()` function clears the destination `VARIANT` parameter before copying anything to it. Therefore, the destination parameter passed to `VariantCopy()` must be cleared first as well. The code in Figure 37demonstrates a vulnerable condition with the same exploitability constraints as the previous example.

Similar problems also exist in other `VARIANT` API functions, most notably the

`VariantChangeType()`/`VariantChangeTypeEx()` functions. These functions will use `VariantClear()` in some but not all conversion cases. The rules for when `VariantClear()` is called on the destination value are for the most part intuitive; they occur when: An invalid conversion attempt is not encountered (i.e., not converting between two incompatible types), and The `VariantClear()` will not cause problems when the source and destination `VARIANT` are the same, such as converting from a `VT_UNKNOWN` to a `VT_DISPATCH`. In terms of auditing for vulnerabilities, any conversion where the destination parameter is uninitialized should be viewed critically.

For example, consider the code in Figure 38. Here we see a similar construct to the previous examples, except this time using `VariantChangeType()`. The following requirements

```
HRESULT MyFunc(IStream* pStream)
{
        VARIANT srcVar;
        VARIANT dstVar;
        IDispatch* pDisp;
        HRESULT hr;

        srcVar.vt = VT_DISPATCH;
        dstVar.vt = VT_DISPATCH;
        hr = pStream->Read(pDisp, sizeof(IDispatch *), NULL);

        if(FAILED(hr)) {
                //VariantClear(&var);
                return hr;
        }
        else {
                srcVar.pdispVal = pDisp;
                hr = VariantCopy(&dstVar, &srcVar);
        }

        return hr;
}
```

Figure 37: Code demonstrating a subtle initialization failure

```
BSTR *ExtractStringFromVariant(VARIANT *var)
{
        VARIANT dstVar;
        HRESULT hr;
        BSTR *res;

        if(var->vt == VT_BSTR)
                return SysAllocString(var->bstrVal);
        else {
                hr = VariantChangeType(&dstVar, var, 0, VT_BSTR);

                if(FAILED(hr))
                        return NULL;
        }

        res = SysAllocString(dstVar.bstrVal);
        VariantClear(&dstVar);

        return res;
}
```

Figure 38: Code sample demonstrating initialization failure with `VariantChangeType()`

for exploitation exist: 1. The destination `VARIANT` is uninitialized, and 2. A conversion from a regular type to `VT_BSTR` will result in `VariantClear()` on the destination `VARIANT` (such as `VT_I4` to `VT_BSTR`). As mentioned previously, successful exploitation of the above vulnerability would require the attacker to be able to influence the stack so that the uninitialized destination `VARIANT` had useful data in it, such as having a type of `VT_DISPATCH` and some sort of valid pointer as the value.

4.3.1 Detecting COM Initialization Failure With Static Analysis

Much like detecting failures in the instantiation of C++ objects is a difficult problem, so too is detecting failures in the instantiation of COM objects. This is an area for future research.

Chapter V

OBJECT LIFETIME PROBLEMS

In the previous chapter, we discussed the sorts of security issues that can arise with complex object types during the instantiation of those objects. In this chapter, we will discuss the sorts of security issues that can arise during the lifetime of the object. We will leave the issues associated with deleting the object to the next chapter.

5.1 C++ Type Confusion

The C++ additions to C create new ways for developers to introduce software vulnerabilities into their code. Some of these extensions introduce safety conditions, including the type confusion error leading to vtable escape vulnerabilities, that compilers cannot identify during code generation. In this section, we discuss these issues, the complexities that C++ introduces into reverse engineering process, and the assumptions that underlie our analyses of Section 5.1.4.

5.1.1 Silent Type Confusion

A number of C++ code-level defects do not present the developer with any sort of compile-time warning. For example, the `static_cast` operator (a) converts a pointer to a base class into a pointer to a derived class, or (b) converts a pointer to or from a `void` pointer. There is no check for object congruence; this is not a language error but an deliberate design choice to allow developers to insert unsafe casts into their software [84]. Casts through `void` pointers clearly deactivate all compiler type-checking for the pointer, but common developer documentation of `static_cast` omits this behavior, presenting only operation (a) [87, 55]. Both cast types (a) and (b) violate type safety, and uses of `static_cast` without additional safety checks by the developer can result in type confusion. Neither Microsoft Visual Studio nor g++ warn of unsafe static casts because to do so would violate the very purpose of the operator. Consider the example code shown in Figure 39a. A human analyst can see that the method debug should never be called on an object of type `class1`. The `static_cast` operation deliberately permits this type of error in software, and both Visual Studio and g++ build the code without warning or error. Running the compiled code will crash, perhaps in an exploitable way. At a low-level, when this code executes, it attempts to dereference the fourth entry in the vtable for `class1`. `Class1`, however, only has two entries in its vtable causing this dereference to read arbitrary memory—a vtable escape bug. This class of vulnerability has impacted widely deployed proprietary software. For example, in March 2010, Microsoft patched a vulnerability in Excel that was the result of C++ object type confusion [65]. In April 2011, Adobe announced a 0-day type confusion vulnerability in their Flash Player [2] after exploits appeared in the wild. Another actively exploited type confusion vulnerability occurred in the Microsoft ATL [64], a set of C++ template code that ships with Visual Studio. Developers were inadvertently including the vulnerable code in their own projects.

5.1.2 Reverse Engineering C++ Software

When C++ code is compiled more high-level information is lost than with what is experienced in C, leading to many unsolved problems in C++ reverse engineering. Dynamic dispatch is one of the most significant challenges: C++ developers can optionally create objects in such a way that the methods of an object are called through indirection. In object-oriented design, polymorphism can be achieved by constructing objects that implement dynamic dispatch. This allows for the substitution of a method's implementation using the same interface. In C++, this is accomplished by declaring a member function `virtual`. All of the virtual functions have a corresponding pointer to the function's implementation in the vtable of the object. They are each stored in the order that that are declared in the object and referenced as an offset from the base of the vtable. For example, consider the code shown in Figure 39a. In this case, there are four virtual member functions in `class2`. When this code is compiled, these four functions appear in the vtable as shown in Figure 40b, and as calls are made to those member functions, they will appear as an indirect call to the base of the vtable plus the offset coresponding to the correct member function.

```
  class class1 {
  public:
      class1();
      ~class1();
      virtual void addRef();
      virtual void print();
      virtual void voidFunc1()  ;
      virtual void debug();

  private:
      int refCount;
  };

  class class2 {
  public:
      class2();
      ~class2();
      virtual void addRef();
      virtual void print();

  private:
      int refCount;
};

  int _tmain(int argc, _TCHAR* argv[])
  {
      class1 C1;
      class2 C2;

      void *pv = &C2;

      C1.addRef();
      C1.print();

      C2.addRef();
      C2.print();

      static_cast<class1*>(pv)->debug();

      return 0;
  }
```

(a) Excerpt of original source (member function implementations omitted)

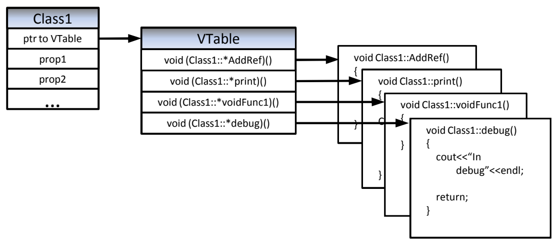
```
.text:00401000 wmain           proc near
.text:00401000
.text:00401000 var_20          = dword ptr -20h
.text:00401000 var_18          = byte ptr -18h
.text:00401000 var_C           = dword ptr -0Ch
.text:00401000 var_4           = dword ptr -4
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 and     esp, 0FFFFFFF8h
.text:00401006                 push    0FFFFFFFFh
.text:00401008                 push    offset loc_401A80
.text:0040100D                 mov     eax, large fs:0
.text:00401013                 push    eax
.text:00401014                 sub     esp, 14h
.text:00401017                 mov     eax, __security_cookie
.text:0040101C                 xor     eax, esp
.text:0040101E                 push    eax
.text:0040101F                 lea     eax, [esp+24h+var_C]
.text:00401023                 mov     large fs:0, eax
.text:00401029                 lea     ecx, [esp+24h+var_18]
.text:0040102D                 call    class1__class1
.text:00401032                 lea     ecx, [esp+24h+var_20]
.text:00401036                 mov     [esp+24h+var_4], 0
.text:0040103E                 call    class2__class2
.text:00401043                 lea     ecx, [esp+24h+var_18]
.text:00401047                 mov     byte ptr [esp+24h+var_4], 1
.text:0040104C                 call    class1__addRef
.text:00401051                 lea     ecx, [esp+24h+var_18]
.text:00401055                 call    class1__print
.text:0040105A                 lea     ecx, [esp+24h+var_20]
.text:0040105E                 call    class1__addRef
.text:00401063                 lea     ecx, [esp+24h+var_20]
.text:00401067                 call    class2__print
.text:0040106C                 mov     eax, [esp+24h+var_20]
.text:00401070                 lea     ecx, [esp+24h+var_20]
.text:00401074                 call    dword ptr [eax+0Ch]
.text:00401077                 xor     eax, eax
.text:00401079                 mov     ecx, [esp+24h+var_C]
.text:0040107D                 mov     large fs:0, ecx
.text:00401084                 pop     ecx
.text:00401085                 mov     esp, ebp
.text:00401087                 pop     ebp
.text:00401088                 retn
.text:00401088 wmain           endp
```

(b) Compiled Binary

Figure 39: C++ code with a type-safety violation

```
.rdata:00402138 off_402138      dd offset sub_4010D0
.rdata:0040213C                  dd offset sub_4010A0
.rdata:00402140                  dd offset nullsub_1
.rdata:00402144                  dd offset sub_4010B0
.rdata:00402148                  dd offset dword_402274
.rdata:0040214C off_40214C      dd offset sub_4010D0
.rdata:00402150                  dd offset sub_4010E0
.rdata:00402154                  align 8
.rdata:00402158                  db  48h ; H
.rdata:00402159                  db    0
.rdata:0040215A                  db    0
.rdata:0040215B                  db    0
.rdata:0040215C                  db    0
```

(a) Disassembly of the vtables for `class2` and `class1`



(b) Structure of a C++ object after compilation

Figure 40: Compiled objects in binary code

5.1.3 Assumptions

Our analyses and implementation provide automated tools that reduce the manual labor a reverse engineer must employ to better understand commercially available software. This may be needed to evaluate the security of a given program or to try to inter-operate with closed source software (e.g., creating a Microsoft Word rendering engine). The techniques presented in this dissertation are designed to analyze legitimate applications like these examples, and they may not have applicability to malware or other obfuscated programs. From a security perspective, legitimate, closed-source applications often require similar levels of analysis as required for malware because attackers will very often leverage software vulnerabilities in legitimate applications as a way to deliver malware payloads. Hence, understanding these vulnerabilities is also extremely important. The binaries that we analyze throughout the paper were all compiled using Microsoft Visual Studio. We chose this as the target compiler for our analysis because most Windows programs that require reverse engineering are built in this environment. In contrast, applications that are built with the GNU developer tools are also usually open source and do not require the complex binary analyses we present here. However, all of the compiler-based issues we discuss are also present in g++, and our analyses could be extended to other platforms. The basis for the issues we discuss are rooted in the C++ standard rather than the implementation of that standard by the various compilers. The analyses presented in this work do not make use of any sort of Runtime Type Information (RTTI). As RTTI is only optionally compiled into the binary, we wanted to ensure our analysis would work in its absence.

5.1.4 Detecting C++ Type Confusion With Static Analysis

In this section we introduce a static data flow analysis known as Object Reaching Definition Analysis. This analysis allows for the identificaiton of vtable escape vulnerabilities by statically analyzing compiled code. The analysis scans through the compiled binary looking for uses of C++ objects, and comparing those uses to the type of object that is pointed to at the use point.

5.1.4.1 Object Reaching Definition Analysis

As C++ code is compiled, it introduces constructs into the binary that make reverse engineering difficult. Notably, object methods declared as virtual introduce layers of indirection that can be nearly impossible to manually traverse. To assist in the analysis of C++ compiled code, we have created a data-flow analysis called Object Reaching Definition Analysis. The goal of this analysis is to resolve the indirect virtual function calls present in binary code. With this resolution in place, analysts can now much more easily navigate the compiled code. Additionally, many automated analyses require a complete static call graph. In resolving the virtual function calls, the static call graph becomes more accurate. Object Reaching Definition Analysis is detailed in the following subsections.

Object Identification In the data flow analyses that will be introduced in the following subsections, we will need to be able to identify all of the new objects that are instantiated in a given basic block. When working directly with source code, identifying the instantiation of objects is rather easy. They are either declared on the stack as with any other sort of variable, or they are instantiated on the heap using the `new` operator. When analyzing binary code, we do not have access to these obvious identifiers and must find other ways to locate the instantiation of new objects. Here we present four heuristics to detect the four ways an object can appear in binary code. Consider the code in Figure 39a. In this code, the objects `C1` and `C2` are declared on the stack. Additionally, Visual Studio is set to aggressively inline functions, so the constructors are inlined as shown in Figure 39b. In this case, we make the assumption that if we encounter a structure where the first element is a pointer to an array of function pointers that we are dealing with an object. In the example in Figure 39b, the first element of a structure is being initialized to a pointer to an array of function pointers at 0x02A. We now assume that this is an object and we will track this unique type throughout the rest of the program. The next sort of object instantiation we have to handle is when the object is declared on the heap using the `new` operator and the constructor is inlined. In this case, we can be more precise in our determination that we are dealing with an object and not just a generic structure. We gain this precision because of the fact that Visual Studio always applies the mangled name `YAPAXI()` to the `new` operator. g++ similarly always applies the mangled name `Znwjxxx()` (where xxx may vary) to the `new` operator. Now we can apply the same logic as above, but only in cases where the pointer to an array of function pointers is being assigned to the value returned by one of the known `new` operators. The third and fourth types of object instantiation that we must be concerned with are when the objects are declared on the stack or heap, and the constructors are not inlined. In this case, we must employ an inter-procedural analysis to determine that we are dealing with an object. For this analysis, we make another assumption. Compilers will nearly always make the call to an object's constructor immediately after the call to `new`. With that, we assume that the next call we encounter after a call to `new` is a constructor, and can validate that assumption with the heuristics mentioned regarding the pointer to a table of function pointers.

Constructor Analysis In the previous section, we loosely covered the hueristic that we use to determine that we are dealing with a constructor. To properly track an object's use throughout the control flow of a program, we need to know more details about that object. These details can be gleaned from the constructor. Specifically, we need to know the size of the vtable, and which function pointers it stores. Additionally, we need to know the number and size of the properties stored within the object. To gain a full understanding of the vtable, we follow the pointer that is assigned to the first element of the object's structure. Figure 40a shows the vtable for an object of type `class1` as referenced in Figure 39a. We can see in the code in Figure 39b that the vtable starting at 0x138 is being referenced by the constructor for `class1`. In order to perform our later analyses, we need to understand the precise size of the vtable and the function pointers contained in that table. Here we will implement some more hueristics. First we need to determine the length of the vtable. We use a few hints to find the end of a vtable for a given object. The first is that if a data element is pointed to by another program point, it is likely the start of some other data structure (i.e., the next element past the end of our vtable). The element at 0x14C is an example of this case. The second hint is that if a pointer in the table does not point to a function, we assume that it is the next element past the end of the table. The element at offset 0x148 is an example of this case. The thrid hint is zero padding. Compilers will often pad the end of a data structure with zeros. Beginning at offset 0x154, we can see zero padding past the vtable for `class2`.

5.1.4.2 Reaching Definition Analysis Algorithm

Now that all of the object instantiations have been identified, we can begin our analysis. The basis of the analyses we will conduct is what we call Object Reaching Definition Analysis. In short, we perform a fixed-point inter-procedural reaching definition analysis for each object definition we encountered during the previous step. The data-flow equations used in Object Reaching Definition Analysis are defined below. for each basic block:

$$REACH_{IN}[S] = \bigcup_{p \in pred[S]} REACH_{OUT}[p] \tag{1}$$

$$REACH_{OUT}[S] = GEN[S] \cup (REACH_{IN}[S] - KILL[S]) \tag{2}$$

where GEN is the set of objects that were identified as being instantiated using the hueristics listed in Section 5.1.4.1 and $KILL$ is the set of objects that are deleted. In our analysis, objects must be tracked interprocedurally. In those cases, $REACH_{IN}$ at the entry of a function F is equal to $REACH[S]$ at the call site to F from a basic block S.

5.1.4.3 Virtual Function Resolution

Now that we have the sets of objects that reach a given program point, we can use that information to resolve the virtual function calls that appear in the binary. There are several cases that can occur when evaluating the reaching definitions. In the first case, only a single object definition reaches. In the second case, a single object definition or NULL reaches. In the third case, a decidable number of object definitions reach. In the final case, an undecidable number of object definitions reach. We will discuss each of these cases below:

1. *Single Object Definition:* In this case, we can make the safe assumption that only a single object definition reaches a given program point. With this assumption, we can

```
.text:00401000 _wmain          proc near
.text:00401000    push    esi
.text:00401001    push    edi
.text:00401002    push    8
.text:00401004    call    ??2@YAPAXI@Z
.text:00401009    add     esp, 4
.text:0040100C    test    eax, eax
.text:0040100E    jz      short loc_401019
.text:00401010    call    sub_401070
.text:00401015    mov     edi, eax
.text:00401017    jmp     short loc_40101B
.text:00401019
.text:00401019 loc_401019:
.text:00401019    xor     edi, edi
.text:0040101B
.text:0040101B loc_40101B:
.text:0040101B    push    8
.text:0040101D    call    ??2@YAPAXI@Z
.text:00401022    add     esp, 4
.text:00401025    test    eax, eax
.text:00401027    jz      short loc_401032
.text:00401029    call    sub_4010A0
.text:0040102E    mov     esi, eax
.text:00401030    jmp     short loc_401034
.text:00401032
.text:00401032 loc_401032:
.text:00401032    xor     esi, esi
.text:00401034
.text:00401034 loc_401034:
.text:00401034    mov     eax, [edi]
.text:00401036    mov     edx, [eax]
.text:00401038    mov     ecx, edi
.text:0040103A    call    edx
.text:0040103C    mov     eax, [edi]
.text:0040103E    mov     edx, [eax+4]
.text:00401041    mov     ecx, edi
.text:00401043    call    edx
.text:00401045    mov     eax, [esi]
.text:00401047    mov     edx, [eax]
.text:00401049    mov     ecx, esi
.text:0040104B    call    edx
.text:0040104D    mov     eax, [esi]
.text:0040104F    mov     edx, [eax+4]
.text:00401052    mov     ecx, esi
.text:00401054    call    edx
.text:00401056    mov     eax, [esi]
.text:00401058    mov     edx, [eax+0Ch]
.text:0040105B    mov     ecx, esi
.text:0040105D    call    edx
.text:0040105F    pop     edi
.text:00401060    xor     eax, eax
```

(a) Original Source

```
class class1 {
public:
  class1();
  ~class1();
  virtual void addRef();
  virtual void print();
  virtual void voidFunc1() { };
  virtual void debug();
};

class class2 {
public:
  class2();
  ~class2();
  virtual void addRef();
  virtual void print();
};

int internalFunction(void *pv) {

  static_cast<class1*>(pv)->addRef();
  static_cast<class1*>(pv)->print();
  static_cast<class1*>(pv)->debug();

  return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
  class1 *C1 = new class1;
  class2 *C2 = new class2;

  internalFunction((void *)C1);
  internalFunction((void *)C2);

  return 0;
}
```

(b) Compiled Binary

Figure 41: C++ code with heap-declared objects

resolve the function calls made to methods of the object by simply indexing into the vtable based on the offset from the base. For example, in the code in Figure 39a, only a single object definition reaches. In the compiled equivalent in Figure 39b, we can see at line 0x078 there is a call to edx. Since we have reconstructed the vtable for the object, we can tell that it is actually a call to `debug()`. When this situation is encountered, as we resolve the virtual function calls, we can also check for "congruence." The concept of congruence is explained in detail in Section 5.1.4.4.

2. *Single Object Definition with NULL:* Visual Studio commonly adds a check for the return from the `new` operator and sets it to NULL in the case of failure. In these instances, it is possible for a developer to end up with a NULL pointer dereference even though they did not explicitly add this code. The code in Figure 41a at line 0x025 is an example of this check. In these cases, our analysis treats the object definition exactly as in case 1. Futher work can be done here to determine the safety of this check insertion.

3. *Decidable Number of Object Definitions:* In this case, multiple object definitions reach a given program point, but we are able to determine the exact number and type of those objects. When this situation is encountered, we first check for congruence between all of the objects that reach. If they are not all congruent, there is likely a type safety condition. We trigger a warning to this condition. If they are congruent, we continue the analysis to determine the safety of the use of the object as done in case 1.

4. *Undecidable Number of Object Definitions:* This case occurs when objects are instantiated and stored in a manner that does not allow our hueristics to determine their type. This scenario is typically encountered when a class pointer is stored in a collection of some type. In other words, when a class pointer is stored and retrieved from the heap in a manner other than direct variable assignment, our reaching definition analysis will fail. An example of this would be instantiating an object and storing its pointer in a `std::map`. Since this object can now be referenced by the key value, we cannot check

for congruence or resolve the virtual function calls.

5.1.4.4 Object Congruence

In addition to using object reaching definition analysis for virtual function resolution, we can also test for "object congruence." We define two objects as being congruent when they are made up of the same number of methods and properties. Additionally, the method at each equivalent location in the two vtables must require the same number and type of arguments. The properties of the two objects must also correspond in size and type at each offset in the property table. As binary code is analyzed in the manner described in Section 5.1.4.3, when an object use is identified, we implement a congruence check to make sure the object is being referenced in a way that is safe with regard to the actual object type that reaches that use. This becomes particularly important when more than one object definition reaches a given use. We have to ensure that all of the object types are congruent, and that any object, when referenced, will be done so safely. If even one of the possible reaching object definitions is unsafe for use, the program point as a whole has to be marked unsafe. This is because static analysis cannot guarantee which object definition will be actually used at runtime.

5.1.4.5 Implementation

In order to test and verify the data flow algorithms presented in Section 5.1.4.2, we created a framework called `llvm-decomp` . This tool allows us to reverse x86 compiled binaries into the intermediate representation used by the LLVM compiler framework. Then, we are able to use the analysis capabilities in LLVM to implement the algorithms for Object Reaching Definition Analysis, virtual function resolution, and object congruence testing. This section presents the details on how this system was built.

5.1.4.6 High-Level Architecture

As compiled binaries are translated into the LLVM IR and analyzed, the data traverses several tools and formats. In this section we describe the high-level architecture of the system as a whole. The diagram provided in Figure 42 shows the process described below.

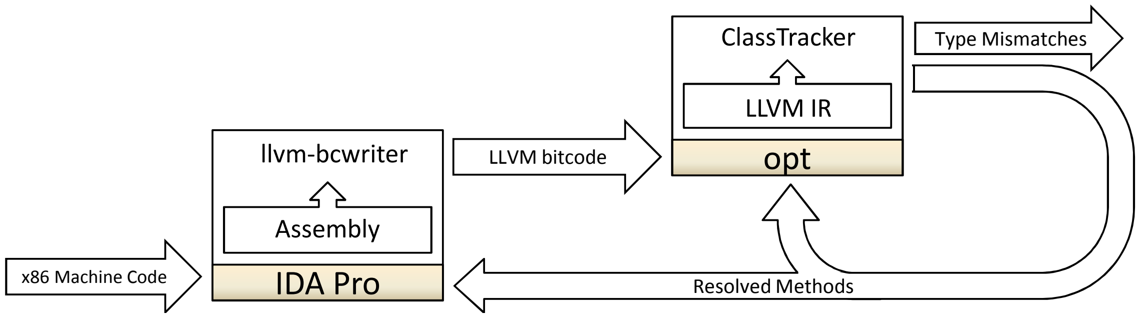


Figure 42: High-Level Architecture of llvm-decomp: Representation of how data flows from x86 assembly through to the LLVM IR for analysis

The initial input into llvm-decomp is C++ compiled x86 machine code. We chose to perform our analyses on compiled code to assist reverse engineers in understanding the complexities that are introduced by C++. There are several commercially available and free tools for disassembling machine code into its assembly equivalent. In `llvm-decomp`, we use IDA Pro for this step. IDA Pro is a commercially available disassembler with an extensive set of tools available for use by reverse engineers. It focuses almost entirely on analyzing the assembly representation of the compiled code. IDA Pro offers a plugin infrastructure whereby a developer can interoperate with the analysis framework. In `llvm-decomp`, we created a plugin for IDA Pro called `llvm-bcwriter`. `llvm-bcwriter` traverses the assembly code and translates it into the LLVM intermediate representation. The converted IR is then written to LLVM's bitcode format to be consumed by their analysis framework in the next step. This conversion process is described in detail below. With a completed bitcode file, we can now implement the analyses detailed in Section 5.1.4.1 using the tool suite provided with LLVM. Specifically, we use a tool called `opt` that allows developers to test custom code analyses. It provides the same infrastructure that would be available inside the compiler without requiring compiled output. opt disassembles the bitcode into the LLVM IR and provides interfaces for analysis. There are several benefits to operating with the LLVM IR because it is a static single assignment form. One of which is the use-def chains are implicit which greatly assists in the reaching definition analysis as described in Section 5.1.4.7. Much like IDA Pro, `opt` provides a plugin infrastructure for custom analysis. In this phase of our framework, we created a plugin for `opt` called ClassTracker. ClassTracker performs a reaching definition analysis on objects as they appear in the LLVM IR.With this reaching definition analysis completed, we can resolve virtual function calls, and perform type-safety checks on object usage. These processes are covered in detail in Section 5.1.4.9. With the virtual function resolution performed by ClassTracker, we can now propogate that information back into the LLVM IR or even the disassembly. With this information provided in these lower-level formats, other analyses are now possible that were previously broken by dynamic dispatch.

Phases of Decompilation The decompilation performed by `llvm-decomp` is completed in multiple passes over the disassembly provided by IDA Pro. This is largely due to some nuances in the LLVM framework as well as in IDA Pro. The phases required are:

1. Create a generic class object: We have to first create a generic class object that can be used as a type specifier for any object we encounter in later phases of decompilation. This class is generated by creating an LLVM global variable consisting of a structure with two pointers. The first points to an arbitrarily large array of function pointers. This will hold the vtable for real objects that are discovered during decompilation. The second points to an arbitrarily large array of unsigned integers. This is used to hold the object properties. Types can be manipulated at a later time.

2. Collect all the functions in the module: In order to properly populate an LLVM module, we have to first collect all of the functions that are contained in the compiled binary. This information is provided to us by IDA Pro using the `getnfunc()` API. This allows us to loop through all of the known functions in IDA Pro, and create the equivalent in the LLVM module. During this process, we must also properly define the number and type of arguments and local variables used in each function. While this information is conveyed graphically in IDA Pro, it is not readily available through the API. As such, we developed a number of heuristics to uncover this information. In the case of C++ compiled binaries, Visual Studio will make frequent use of `thiscall` and `fastcall` calling conventions. This adds to the number of heuristics required to properly define the functions in the LLVM module.

3. Insert basic blocks into each function: Once the functions are defined in the LLVM module, we can add the basic blocks to the function. Basic block information is not provided through the base API in IDA Pro. As such, we implemented our own basic block discovery algorithm based on the one by Moretti, et al [68].

4. Add instructions to basic blocks: Now that the basic blocks for each function are defined, we can add the instructions. At this point we need to reduce the assembly equivalent to the LLVM intermediate representation. This phase is where the majority of the work is required to complete the binary translation. The following steps are followed:

    (a) Survey all available LLVM IR operations: First we need to know all of the operations that we may need to represent in the IR. The LLVM IR instruction set is rather small and well documented.

    (b) Enumerate collections of assembly instructions: We then enumerate all of the collections of assembly instructions that map to the appropriate LLVM instruction. This again is represented as a finite set of collections of instructions that have the same functional meaning as one of the higher-level LLVM operations.

    (c) Track processor state for a given basic block: During the binary translation process, the processor state can differ wildly depending on which branch of the control flow is taken. As such, a processor state object is maintained for each basic block. In this object, we store the state of the registers and the stack, so as data is referenced, we have an accurate representation of what data is held at that program point.

    (d) Insert phi nodes into the IR: We have to be able to account for conditions where a register or memory location can differ in value depending on the control flow leading to a given program point. To account for this condition, the LLVM IR implements phi nodes. Phi nodes are operators that allow for a single variable to represent two or more values. As phi nodes are encountered, the analysis proceeds as if that variable is equal to more than one value at the same time.

(e) Complete instruction mapping: Now that we can track processor state, and handle multiple control flows, we can complete the mapping from x86 assembly to LLVM IR. This is handled with a rather straightforward state machine in which we track the known groups of instructions that match to the known LLVM operators. These groups of instructions can of course be interleaved in the assembly, so often several instruction groups are tracked at once.

(f) Handle unknown instructions: It is unrealistic to plan for all possible collections of instructions ahead of time. As such, there are times when the instruction collection states get "lost." In these cases, LLVM allows for the insertion of platform-specific inline assembly. When llvm-bcwriter cannot find a mapping between a collection of assembly instructions and an LLVM operation, it will emit the x86 assembly inline into the IR. The processor state object is updated to reflect the changes made in the assembly and the binary translation continues. Under some circumstances, the binary translation is able to continue without any other errors. In other cases, the remaining binary code was too dependent on the instructions that were not understood and the rest of the translation fails.

5. Emit LLVM binary representation: Once the process listed above is complete, we have an LLVMmodule which represents the binary equivalent. At this point, the code can be written to the LLVM binary form which they call bitcode (note - not bytecode). The LLVM module that has been created is written to bitcode using the `WriteBitcodeToFile()` LLVM API. This will generate the LLVM version of a compiled binary and write it to disk. This binary, however, is a binary representation of the LLVM SSA-based intermediate representation; not a native executable. With a completed LLVM intermediate representation of the code, we can now move on to use the LLVM analysis tools to implement the analyses listed in Section 5.1.4.2. However, it is worth noting that there is a benefit for a reverse engineer at this point in the framework. The SSA-based intermediate reprsentation used by LLVM is very easy to read, and in many cases will represent vulnerable code more clearly than the x86

assembly equivalent. As such, in the `llvm-decomp` framework, we run `llvm-dis` to generate the textual equivalent of the intermediate representation and write the results to disk. We are then able to load this file in a new view in IDA Pro to provide two different representations of the same binary to assist in analysis.

5.1.4.7 Analyzing the Intermediate Representation

The next phase of llvm-decomp is provide the bitcode file that was generated by llvm-bcwriter as input into the tool opt. opt allows us to create custom analyses while taking advantage of all of the infrastructure available in the LLVM compiler.

5.1.4.8 Interoperating with LLVM

`opt` allows for the creating and testing of custom compiler analyses without the need for generating compiled output (although it can do that too). This is done by loading a custom library by passing the `-load` option to `opt`. This library then registers to be called at various points in the analysis. It can be called at the module, function, basic block, or instruction levels. At each of these points, the function being called can assert its placement in the dependency tree. In other words, the custom analysis can be called before or after any of the pre-existing analyses. At each of those analysis levels, a final hook point is available called doFinalization. This allows for a set of "clean up" routines to be run after all of the analysis is complete.

5.1.4.9 ClassTracker

In the `llvm-decomp` framework, we created a plugin for `opt` called ClassTracker. ClassTracker performs all of the analysis that is described in Section 5.1.4.2. To accomplish each of these analyses, it performs the following tasks:

1. Collect all constructors: ClassTracker first registers for a function-level analysis to collect all of the constructors it can find in the IR. This is mostly important for the cases where the constructor is not inlined. In cases where the constructor is inlined it would be simple to combine this step with the next step. When the constructor is explicitly called, this analysis works better with a known set of constructors ahead of

time. Each constructor is stored as an opaque value that we can later reference as the object's type. Without any other sort of type identification in the binary, this opaque value will suffice. An object is created of the stored type using the constructor analysis techniques detailed in Section 5.1.4.1.

2. Identify all new objects: In the next pass over the code, ClassTracker again registers for a functionlevel analysis and enumerates all of the instantiations of new objects. This is done using the hueristics presented in Section 5.1.4.1. Information about the object is stored for later use. This includes the instruction that instantiated the object, and the opaque value that represents its type.

3. Identify object use: ClassTracker then registers for a doFinalization analysis at the module level. In this analysis, the code is scanned looking for all object uses. This is done by looking for dereferences from a base pointer. This rather loose heuristic works just fine for the reasons explained in the next point.

4. Perform reaching definition analysis: This analysis step is to perform a reaching definition analysis on the object definitions that reach the program points identified in the previous step. In LLVM, this is extremely simple. Because the LLVM IR is an SSA-based IR implementing phi nodes, the reaching definition analysis is basically complete. In an SSA format, the use-point of a given object must chain directly back to the definition-point of that object. Consider Appendix A, for example. Here we can see the instruction call %35. Because of the SSA format, we can follow this definition back to %34, %33, and so on until we hit the phi node at %11, and can determine the class type. Additionally, if more than one definition reaches at the entry point to a given basic block, this information would be captured in a phi node. Therefore the phi node is actually $\text{REACH}_{\text{IN}}[\text{S}]$ from data flow equation (1) in Section 5.1.4.2 So following each path back from this point will uncover the various definitions that reach. In cases where step 3 incorrectly identified an object use, there will be no object definitions that reach, and this instruction is discarded.

5. Perform virtual function resolution: Now that we have collected all of the object use

points and their potential definitions, we can fix up the virtual function calls. This is done by iterating over each of the instructions found in step 4. For each call instruction, the offset from the base of the object is determined and used to index into the vtable of the object stored in step 2. This lookup provides the function name (or address) that is actually being referenced for that object type. If more than one object definition reaches, this lookup occurs for each object type.

6. Perform congruence check: As the function lookups are occuring the previous step, we simultaneously check to make sure that there actually is a function at the dereferenced offset for the any and all object definitions that reach that instruction. In cases where the code attempts to dereference a function pointer beyond the end of the vtable, the analysis reports an error.

7. Propagate information back: Now that the virtual function calls have been resolved, we can propagate that information back into IDA Pro and the LLVM IR. In both cases, the function name (or address) is retreived using the procedure in step 5, and inserted into the text of the assembly or IR as a comment next to its use. In the case of IDA Pro, it recognizes this comment as being a function name (or address) and allows the user to navigate to that function by highlighting the comment and pressing enter.

5.1.4.10 Results

By employing the data flow analysis techniques documented in this dissertation, we demonstrate that we can increase the effectiveness of existing static analysis techniques on compiled C++ code, as well as identify a class of vulnerability that is often overlooked by existing techniques. Most importantly, all of this analysis can be performed on a compiled binary with no access to the original source. This allows for third parties like software developers, security analysts, or a software consumer software looking to validate code quality to gain a much better view into the code constructs embedded in a compiled binary. To test our decompilation framework, we created test programs each representing one of the four

```
class class1 {
public:
    class1();
    ~class1();
    virtual void addRef();
    virtual void print();
    virtual void voidFunc1() { };
    virtual void debug();
};

class class2 {
public:
    class2();
    ~class2();
    virtual void addRef();
    virtual void print();
};

int _tmain(int argc, _TCHAR* argv[]) {
    class1 *C1 = new class1;
    class2 *C2 = new class2;

    void *pv = C2;

    C1->addRef();
    C1->print();

    C2->addRef();
    C2->print();

    static_cast<class1*>(pv)->debug();

    return 0;
}
```

Figure 43: Original Source

```
.text:00401000 _wmain proc near
.text:00401000  push esi
.text:00401001  push edi
.text:00401002  push 8
.text:00401004  call ??2@YAPAXI@Z
.text:00401009  add esp, 4
.text:0040100C  test eax, eax
.text:0040100E  jz short loc_401019
.text:00401010  call sub_401070
.text:00401015  mov edi, eax
.text:00401017  jmp short loc_40101B
.text:00401019
.text:00401019 loc_401019:
.text:00401019  xor edi, edi
.text:0040101B
.text:0040101B loc_40101B:
.text:0040101B  push 8
.text:0040101D  call ??2@YAPAXI@Z
.text:00401022  add esp, 4
.text:00401025  test eax, eax
.text:00401027  jz short loc_401032
.text:00401029  call sub_4010A0
.text:0040102E  mov esi, eax
.text:00401030  jmp short loc_401034
.text:00401032
.text:00401032 loc_401032:
.text:00401032  xor esi, esi
.text:00401034
.text:00401034 loc_401034:
.text:00401034  mov eax, [edi]
.text:00401036  mov edx, [eax]
.text:00401038  mov ecx, edi
.text:0040103A  call edx
.text:0040103C  mov eax, [edi]
.text:0040103E  mov edx, [eax+4]
.text:00401041  mov ecx, edi
.text:00401043  call edx
.text:00401045  mov eax, [esi]
.text:00401047  mov edx, [eax]
.text:00401049  mov ecx, esi
.text:0040104B  call edx
.text:0040104D  mov eax, [esi]
.text:0040104F  mov edx, [eax+4]
.text:00401052  mov ecx, esi
.text:00401054  call edx
.text:00401056  mov eax, [esi]
.text:00401058  mov edx, [eax+0Ch]
.text:0040105B  mov ecx, esi
.text:0040105D  call edx
.text:0040105F  pop edi
.text:00401060  xor eax, eax
```

Figure 44: Corresponding Binary

```
"401034":                ; preds = %"401032", %"401029"
        . . .
        %25 = getelementptr %0* %24, i32 0, i32 0 ; <i32 (...)***>
        . . .
        call void %29()
        . . .
        %31 = getelementptr %0* %30, i32 0, i32 0 ; <i32 (...)***>
        . . .
        call void %35()
        . . .
        %37 = getelementptr inbounds %0* %36, i32 0, i32 0 ; <i32 (...)***>
        . . .
        call void %41()
        . . .
        %43 = getelementptr inbounds %0* %42, i32 0, i32 0 ; <i32 (...)***>
        . . .
        call void %47()
        . . .
        %49 = getelementptr inbounds %0* %48, i32 0, i32 0 ; <i32 (...)***>
        . . .
        call void %53()
        . . .
        br label %return
```

Figure 45: Excerpt from LLVM bitcode file generated by llvm-bcwriter

combinations of stack or heap object declaration and inline or explicit constructor. These programs were compiled without symbols and were provided to IDA Pro as input. In each case, the system was tested for its ability to resolve virtual function calls and to identify instances of type confusion. An example of this process is detailed in Section 5.1.4.10. We created a more complex test where an object is declared in one function and referenced in another function, testing the ability of llvm-decomp to perform the analyses described in Section 6.1.3.1 on an interprocedural basis. This test is detailed in Section 5.1.4.10.

Heap-Declared Object, Explicit Constructor Figure 43 shows an unsafe use of static cast through a void pointer resulting in a vtable escape error. The source code shows two heap-declared objects `C1` and `C2`. In the disassembly of the code shown in Figure 44, we see that the constructors for the two objects are explicitly called at lines 0x010 and 0x029 respectively. Additionally, we can see several calls to virtual functions. Lines 0x03A and 0x043 correspond to the calls to `addRef()` and `print()` for `C1`. Lines 0x04B and 0x054 correspond to the calls to `addRef()` and `print()` for `C2`. We can see the erroneous call to debug at line 0x05D. After translating the code in Figure 44 to the LLVM IR, it appears as shown in Figure 45. This is an excerpt of the full code that is emitted from `llvm-bcwriter`. In the LLVM IR, we can see the virtual function calls just as in the disassembly, but it is still just as unclear which function will actually get called. In the LLVM IR, we see the instructions `call void %29` and `call void %35`, which correspond to the calls to `addRef()` and `print()` for `C1`. We also see the instructions `call void %41` and `call void %47`, corresponding to the same functions in `C2`. The final call, `call void %53`, corresponds to the erroneous call to debug. When the Object Reaching Definition analysis in ClassTracker is applied to this code, the various function calls were resolved except for the call to debug. In this case there was no corresponding method in the vtable, and ClassTracker returns a vtable escape error. The detailed output from ClassTracker can be seen in Appendix B.

Interprocedural Analysis The previous example shows how `llvm-decomp` is able to translate binary code into the LLVM IR and perform Object Reaching Definition Analysis. However, the vulnerable code that it was able to identify is very unlikely to ever appear in production code. We would hope that this sort of type-casting would be identified through simple code review. Consider the more complex example in Figure 46: an object is declared in one function and referenced by another function. This code is much more realistic, especially when we consider the possibility that the function `internalFunction()` could be called from many different places in the code, each passing in different object types. In this example, the source code shown in Figure 46 is compiled to the binary representation shown in Figure 47.

Sub 401000 corresponds to the function `internalFunction()`. We can see in this subroutine that it makes three virtual function calls, corresponding to `addRef()`, `print()`, and `debug()`. This binary is provided as input into `llvm-bcwriter`, which produces the LLVM IR shown excerpted in Figure 45a. Here we can see the calls to the virtual functions appearing as `call void %6`, `call void %13`, and `call void %20` respectively. When running ClassTracker on the IR shown in Figure 46, the analysis made two passes over the function internalFunction because it is called twice in this code. In the first pass, the virtual function calls were resolved as they belong to class `C1`. In the second pass, we see a second function

```
class class1 {
public:
        class1();
        ~class1();
        virtual void addRef();
        virtual void print();
        virtual void voidFunc1() { };
        virtual void debug();
};

class class2 {
public:
        class2();
        ~class2();
        virtual void addRef();
        virtual void print();
};

int internalFunction(void *pv) {
        static_cast<class1*>(pv)->addRef();
        static_cast<class1*>(pv)->print();
        static_cast<class1*>(pv)->debug(); return 0;
}

int _tmain(int argc, _TCHAR* argv[]) {
        class1 *C1 = new class1;
        class2 *C2 = new class2;
        internalFunction((void *)C1);
        internalFunction((void *)C2);
        return 0;
}
```

Figure 46: Original source for definition of `class1`

```
.text:00401000 sub_401000 proc near
.text:00401000
.text:00401000  mov eax, [esi]
.text:00401002  mov edx, [eax]
.text:00401004  mov ecx, esi
.text:00401006  call edx
.text:00401008  mov eax, [esi]
.text:0040100A  mov edx, [eax+4]
.text:0040100D  mov ecx, esi
.text:0040100F  call edx
.text:00401011  mov eax, [esi]
.text:00401013  mov edx, [eax+0Ch]
.text:00401016  mov ecx, esi
.text:00401018  call edx
.text:0040101A  xor eax, eax
.text:0040101C  retn
.text:0040101C sub_401000 endp
.text:0040101C
.text:00401020 _wmain proc near
.text:00401020  push esi
.text:00401021  push edi
.text:00401022  push 8
.text:00401024  call ??2@YAPAXI@Z
.text:00401029  add esp, 4
.text:0040102C  test eax, eax
.text:0040102E  jz short loc_401039
.text:00401030  call sub_401080
.text:00401035  mov esi, eax
.text:00401037  jmp short loc_40103B
.text:00401039
.text:00401039 loc_401039:
.text:00401039  xor esi, esi
.text:0040103B
.text:0040103B loc_40103B:
.text:0040103B  push 8
.text:0040103D  call ??2@YAPAXI@Z
.text:00401042  add esp, 4
.text:00401045  test eax, eax
.text:00401047  jz short loc_401061
.text:00401049  call sub_4010C0
.text:0040104E  mov edi, eax
.text:00401050  call sub_401000
.text:00401055  mov esi, edi
.text:00401057  call sub_401000
.text:0040105C  pop edi
.text:0040105D  xor eax, eax
.text:0040105F  pop esi
.text:00401060  retn
.text:00401061
.text:00401061 loc_401061:
.text:00401061  xor edi, edi
```

Figure 47: Compiled binary for `class1`

pointer assigned to each call site. These are the methods belonging to class `C2`. Here again ClassTracker reported for the erroneous call to debug.

5.2 COM Type Confusion

As we previously saw in the technology overview of this dissertation, the `VARIANT` data structure is used extensively throughout Microsoft code as a standardized, language agnostic method of representing a variety of data types. The API for manipulating `VARIANT` data structures has been introduced in the overview section of this dissertation. We will now explore how mismanagement of `VARIANT` structures either directly or through the well-defined API can lead to a number of subtle type confusion scenarios.

5.2.1 `VARIANT` Type Confusion Attacks I: Permissive Property Maps

As was discussed earlier, Microsoft's ATL helps developers rapidly develop COM components by distributing template code for a collection of interfaces. Microsoft has written the template code in an abstract manner, which allows the template code to be used in a large variety of situations; however, there are also subtle consequences of utilizing some of the available code. Specifically, the manner in which the developer used to specify COM object properties using property maps has some subtle nuances that could potentially lead to opportunities for an attacker to perform type confusion attacks. Consider the following macros available in version 9 of the Microsoft ATL, which can be used for specifying individual properties within a property map.

It is important to note that neither `PROP_ENTRY` nor `PROP_ENTRY_EX` require a parameter to specify the `VARIANT` type. Recall from our previous discussion about persistence that when these functions are used, the persistence stream will contain two bytes that identify the serialized type preceding the serialized data. Once the member being described has been de-serialized, the ATL code will call the put property method of the `IDispatch` interface that the property map specifies in order to write the data to the COM object. In summary, utilizing these macros provides a possible opportunity to provide any type of `VARIANT` to the put method of the `IDispatch` interface without forcing coercion to a specific data type. If the developer fails to take into consideration that the put method may be supplied with

```
struct ATL_PROPMAP_ENTRY {
        LPCOLESTR szDesc;
        DISPID dispid;
        const CLSID* pclsidPropPage;
        const IID* piidDispatch;
        DWORD dwOffsetData;
        DWORD dwSizeData;
        VARTYPE vt;
};

#define PROP_DATA_ENTRY(szDesc, member, vt) \
        {OLESTR(szDesc), 0, &CLSID_NULL, NULL, \
        offsetof(_PropMapClass, member), \
        sizeof(((_PropMapClass*)0)->member), vt},

#define PROP_ENTRY(szDesc, dispid, clsid) \
        {OLESTR(szDesc), dispid, &clsid, &__uuidof(IDispatch), \
        0, 0, VT_EMPTY},

#define PROP_ENTRY_EX(szDesc, dispid, clsid, iidDispatch) \
        {OLESTR(szDesc), dispid, &clsid, &iidDispatch, 0, 0, VT_EMPTY},

#define PROP_ENTRY_TYPE(szDesc, dispid, clsid, vt) \
        {OLESTR(szDesc), dispid, &clsid, &__uuidof(IDispatch), 0, 0, vt},

#define PROP_ENTRY_TYPE_EX(szDesc, dispid, clsid, iidDispatch, vt) \
        {OLESTR(szDesc), dispid, &clsid, &iidDispatch, 0, 0, vt}
```

Figure 48: Example COM property map

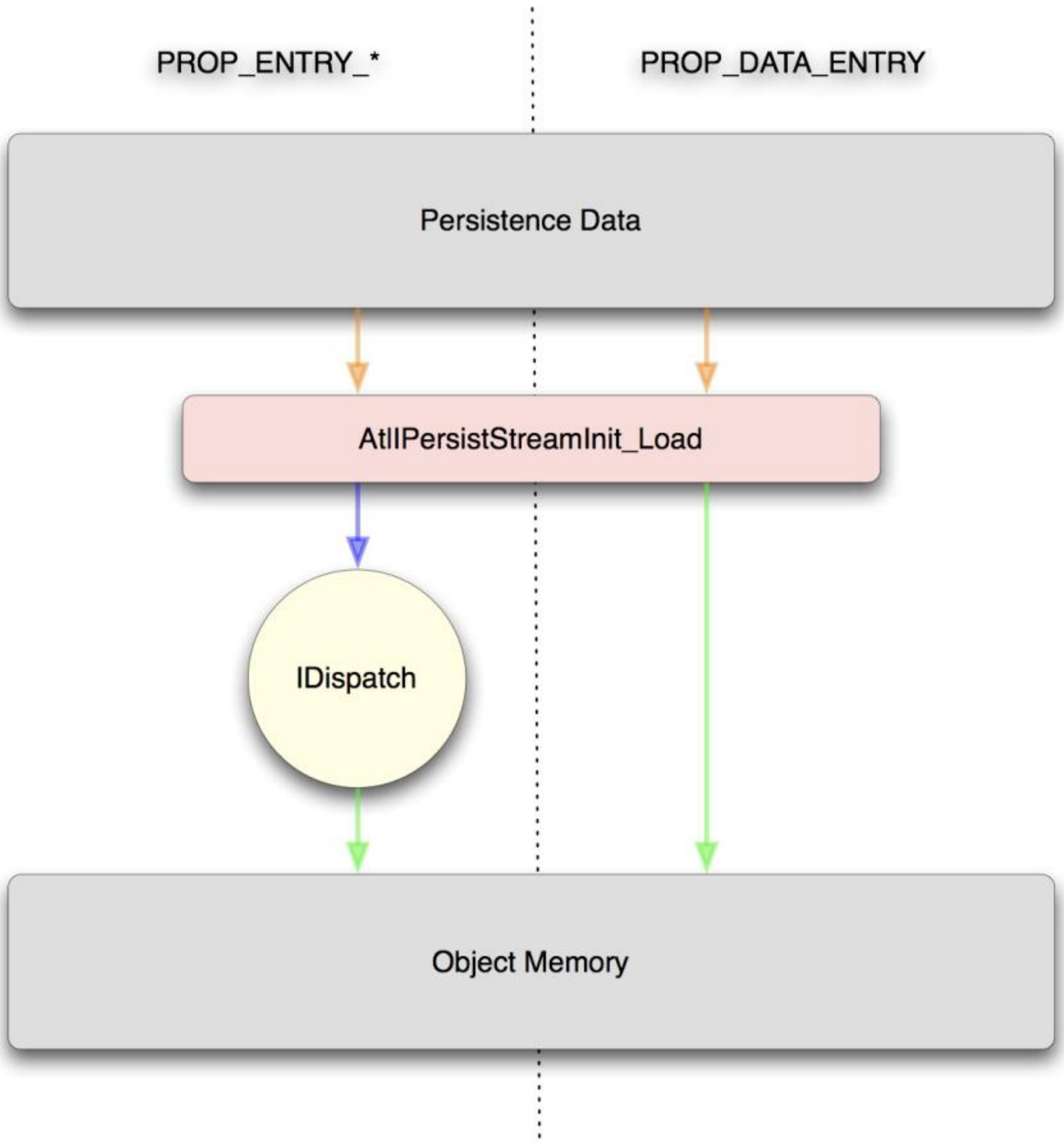


Figure 49: Difference between `PROP_ENTRY_*` macros and `PROP_DATA_ENTRY`

an arbitrary `VARIANT` type, then using this type of property declaration can lead to possible type confusion problems. This type of vulnerability is more likely to be found in objects that aren't used in Internet Explorer, or in interfaces that implement `IDispatch` that are specified in the property map, but are not accessible from Internet Explorer. Developers may also elect to use `PROP_DATA_ENTRY()` instead of `PROP_ENTRY()`. The `PROP_DATA_ENTRY` macro is unique, in that the data for that property is not filtered by an `IDispatch` interface. Instead, it is written directly to an offset within the class memory that holds the property data. If the variant type supplied to the macro is `VT_EMPTY`, then the persistence code will read up to the number of bytes available for the property within the class. The process for unpacking `PROP_DATA_ENTRY` properties versus `PROP_ENTRY` macros is illustrated in Figure 49. So, the use of the `PROP_DATA_ENTRY()` macro provides attackers with two interesting opportunities:

1. The ability to create a property directly in the destination object's memory possibly without having any typing requirements, and

2. The ability to provide properties that have undergone absolutely no validation If the `PROP_DATA_ENTRY` macro is specified in a type-less manager then these properties are quite dangerous.

If they are constructed with the type specified as `VT_EMPTY`, then code that subsequently utilizes such properties will almost certainly contain type confusion vulnerabilities, since it has no way to validate what type of data it is operating on. For example, consider a case where a `PROP_DATA_ENTRY` property is intended to be a pointer to a string or some other more complex object. By specifying an integer type instead of the intended object, a type confusion vulnerability will be triggered, with the end result more than likely being arbitrary execution. Conversely, there may be a situation where a property member is expected to be an integer, but the attacker specifies a pointer instead (by specifying a string or something else). This example type confusion vulnerability will more than likely result in an information leak, and ultimately disclose the value of a pointer. These types of problems are becoming increasingly useful when attempting to bypass memory protection mechanisms

```
#ifndef VT_TYPEMASK
#define VT_TYPEMASK 0xfff
#endif

WXDLLEXPORT bool wxConvertOleToVariant(const VARIANTARG& oleVariant,
        wxVariant& variant) {
        switch (oleVariant.vt & VT_TYPEMASK) {
        case VT_BSTR: {
                wxString str(wxConvertStringFromOle(oleVariant.bstrVal));
                variant = str; break;
        }

        ...
```

Figure 50: Code sample showing a subtle `VARIANT` type confusion vulnerability

found in contemporary Windows operating systems. Furthermore, it is worth considering that `PROP_DATA_ENTRY` properties are set directly, and hence bypass any level of validation that the put property of the `IDispatch` interface may enforce. This means there may be cases where setting these properties directly may circumvent the sanitization process to some degree, since it might be carried out in the put property method. Therefore, there are potential opportunities for an attacker to exploit the object in question when the property is utilized under the tenuous assumption that it is sanitized in a certain manner. `VARIANT`

5.2.2 Type Confusion Attacks II: Misinterpreting Types

One area that is prone to potential problems when dealing with `VARIANT` data structures is correctly interpreting the vt member. In contrast to the NPAPI variant data structures, recall that the type parameter in a `VARIANT` can be a basic type, or a complex type composed of bits that represent a basic type and a modifier (or two modifiers, if one of them is `VT_BYREF`). The misinterpretation of the vt member can occur when bit masking is performed incorrectly, leading to subtle vulnerabilities where the `VARIANT`s value is utilized as one type when it is in fact, another. To illustrate this point, consider the following code:

The astute reader will notice that this code has a very obvious flaw: a type check is performed using a mask to obtain the basic type of the `VARIANT`. In the case of a BSTR, the string is passed to a function which basically duplicates it. The problem here is that if a modifier is used, the `VARIANT` will not contain a BSTR as its value parameter. If the caller

```
SAFEARRAY *psa;
ULONG *pValue

// Test if object is an array of integers
VARTYPE baseType = pVarSrc->vt & VT_TYPEMASK;

if( (baseType != VT_I4 && baseType != VT_UI4) ||
        ((pVarSrc->vt & VT_ARRAY) == 0) )
          return -1;

psa = pVarSrc->parray;

// operate on SAFEARRAY
SafeArrayAccessData(psa, &pValues);

...
```

Figure 51: An attempt at fixing the vulnerable code

of this function were to supply a `VARIANT` with the type (`VT_BYREF|VT_BSTR`) for example, it would cause a pointer to a BSTR to be placed within the `VARIANT` rather than a `BSTR`. (A `BSTR` is really a `WCHAR *` with a 32-bit length preceding it, so a `BSTR *` is a `WCHAR **`.) Therefore, utilization of any modifiers on `VARIANT`s passed to this function will result in a type confusion vulnerability. Consider this slightly more subtle example:

This code performs some checking to ensure that an input type is an array of either signed, or unsigned integers. If it is not, then an error is signaled by returning the value -1. However, there is also a problem in this code – the check for the variant type fails to take into account that the type can have the `VT_BYREF` bits set. Since the `VT_ARRAY` modifier is not mutually exclusive with `VT_BYREF`, the above code has a type confusion vulnerability when dealing with a `VARIANT` with the type (`VT_BYREF|VT_ARRAY|VT_I4`). In this case, a `SAFEARRAY **` will be incorrectly interpreted as a `SAFEARRAY *`, leading to out of bounds memory accesses. The following code is a real-world example taken from IE (all present versions). This example is part of the core marshalling code for the DOM. The code in question is charged with verifying `VARIANT` parameters received from scripting hosts plugged into the DOM are correct and, if necessary, converting those parameters into the expected types. Although each DOM function takes different types of parameters, most marshalling routines, at their core, use the same function, `VARIANTArgToCVar()`, which takes a single `VARIANT` and attempts to convert it to the expected type. The vulnerable code is shown

```
int VARIANTARGToCVar(VARIANT *pSrcVar, int *res,
        VARTYPE vt, PVOID outVar, IServiceProvider *pProvider,
        BOOL bAllocString)
{
        VARIANT var;

        VariantInit(&var);

        if(!(vt & VT_BYREF)) {
                // Type mismatch - attempt conversion

                if( (pSrcVar->vt & (VT_BYREF|VT_TYPEMASK))
                        != vt && vt != VT_VARIANT) {
                        hr = VariantChangeTypeSpecial(&var,
                                pSrcVar, vt, pProvider, 0);

                        if(FAILED(hr))
                                return hr;

                        ... more stuff ...

                        return hr;
                }

                switch(vt) {
                case VT_I2:
                        *(PSHORT)outVar = pSrcVar->iVal;
                        break;
                case VT_I4:
                        *(PLONG)outVar = pSrcVar->lVal;
                        break;
                case VT_DISPATCH:
                        *(PDISPATCH)outVar = pSrcVar->pdispVal;
                        break;
                ... more cases ...
                }
        }
}
```

Figure 52: Attempted fix that ignores the complete length of the type mask

below.

The code in question attempts to retrieve the value of an input parameter, `pSrcVar`, performing a type conversion if the received `VARIANT` isn't of the expected type given in the vt parameter. The problem in this code occurs when comparing the received input `VARIANT`s type with the expected type. Specifically, a test is done by comparing the expected type with the input type after the input type has been masked with (`VT_BYREF|VT_TYPEMASK`), or 0x4FFF. Performing this mask loses significant information, which in this case is the `VT_ARRAY` (0x2000) and `VT_VECTOR` (0x1000) modifiers. To illustrate the problem, consider the case where this function is expecting a VT_DISPATCH input type (0x0009) and the input VARIANT is an array of VT_DISPATCH types (VT_ARRAY|VT_DISPATCH, or 0x2009). Since (0x2009 & 0x4FFF) produces the result 0x0009, or VT_DISPATCH, this code will incorrectly assume it received an IDispatch object rather than an array of IDispatch objects. The result is this function signals success and returns a pointer to a SAFEARRAY which it has incorrectly evaluated as a pointer to an IDispatch interface. Thus, this code culminates to a type confusion vulnerability. An assessor auditing for vulnerabilities in the use of VARIANT type masks must pay close attention to how the vt member of the VARIANT is manipulated. Specifically, the masking of input VARIANT types needs to be performed with caution to ensure that information is not overlooked when performing any validation steps. VARIANT

5.2.3 Type Confusion Attacks III: Direct Type Manipulation

Another construct that can result in type confusion vulnerabilities is directly manipulating the vt member of a VARIANT, rather than using the API functions. Although this should be a fairly straightforward task in theory, subtle vulnerabilities can be introduced by either not correctly enforcing data types, or not correctly ensuring that a type conversion was successful. For example, the following code has been taken from Microsoft's internal version of the ATL. This code is invoked when performing de-serialization of a COM object from a persistence stream. Note that in this particular example, the VARIANT data structure is wrapped in a C++ object, CComVariant. The class member vt in this code corresponds to the vt type variable in a VARIANT structure. The example listed above is contrived; however, the we have identified a real-world scenario where this type of bug has occurred. Microsoft's internal version of the ATL has special code to process variants in a persistence stream that is similar to the following example.

The issue with the above code is that the return value of the ChangeType() function is not checked before manually setting the variant type. This mistake allows an attacker to make the program believe a BSTR value the attacker supplied is any type not handled in the fixed width data types handler. In one scenario, an attacker can specify that a string that he has supplied should be treated as an array of VT_DISPATCH objects. When this function

```
inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
{
        ATLASSERT(pStream != NULL);
        HRESULT hr;
        hr = VariantClear(this);

        if (FAILED(hr))
                return hr;

        VARTYPE vtRead;

        hr = pStream->Read(&vtRead, sizeof(VARTYPE), NULL);

        if (hr == S_FALSE)
                hr = E_FAIL;

        if (FAILED(hr))
                return hr;

        vt = vtRead;

        //Attempts to read fixed width data types here

        CComBSTR bstrRead;

        hr = bstrRead.ReadFromStream(pStream);
        if (FAILED(hr))
                return hr;

        vt = VT_BSTR;
        bstrVal = bstrRead.Detach();

        if (vtRead != VT_BSTR) {
                hr = ChangeType(vtRead);
                vt = vtRead;
        }

        return hr;
}
```

Figure 53: Code excerpt from Microsoft ATL

returns an error, the caller will attempt to free the string using the `VariantClear()` function. This ends up causing the program to treat the attacker supplied string as an array of vtables, a clear type confusion error, ultimately allowing for arbitrary code execution.

5.2.4 Detecting Type Confusion Attacks With Static Analysis

Detecting type confusion attacks against COM objects is an opportunity for future research. A similar analysis could be developed to the one described in Section 5.1.4. While the memory layout for a COM object is not identical to that of a C++ object, it is similar. The concepts from Section 5.1.4 could be adapted to fit the memory layout for COM objects. Further, because COM objects are self documenting, gathering information about the real structure of the object is easier than in the case of C++.

5.3 `VARIANT` *Shallow Copies*

When a `VARIANT` object is duplicated, it is typically done with `VariantCopy()`, but just a simple `memcpy()` is also used in many cases. `VariantCopy()` is the preferred method, since it will do an object-aware copy – if the `VARIANT` being copied is a string, it will duplicate the memory. If the object being copied is an object, it will add a reference count. In contrast, `memcpy()` obviously performs a shallow copy – if the `VARIANT` contains any sort of complex object, such as an `IDispatch`, a pointer to the object will be duplicated and utilized without adding an additional reference to the object. If the result of this duplicated `VARIANT` is retained, the object being pointed to could be deleted, if every other instance of that object is released. The code in Figure 54 demonstrates this vulnerable construct.

There is also a more subtle variation on the attack – this time using `VariantCopy()`. In some ways, `VariantCopy()` can be also be considered a shallow copy operation, in that any `VARIANT` that has the `VT_BYREF` modifier will not be deep-copied; just the pointer will be copied.

Consider the code in Figure 55. This example shows a sample ActiveX property that simply takes a `VARIANT` and stores it, and optionally returns it to the user. The problem with this code is that `VariantCopy()` is used rather than `VariantCopyInd()`. If a `VARIANT` is supplied that has the type (`VT_BYREF|VT_DISPATCH`) for example, a simple pointer copy is

```
HRESULT CMyObject::put_MyProperty(VARIANT src)
{
        HRESULT hr;

        memcpy((void *)&m_MyProperty, (void *)&src, sizeof(VARIANT));

        return S_OK;
}

HRESULT CMyObject::get_MyProperty(VARIANT *out)
{
        HRESULT hr;

        if(out == NULL)
                return E_FAIL;

        VariantInit(out);
        memcpy(out, (void *)&m_MyProperty, sizeof(VARIANT));

        return S_OK;
}
```

Figure 54: Code demonstrating a shallow copy of an object

```
HRESULT CMyObject::put_MyProperty(VARIANT src)
{
        HRESULT hr;
        VariantInit(&m_MyProperty);

        hr = VariantCopy(&m_MyProperty, &src);

        if(FAILED(hr))
                return hr;

        return S_OK;
}

HRESULT CMyObject::get_MyProperty(VARIANT *out)
{
        HRESULT hr;

        if(out == NULL)
                return E_FAIL;

        VariantInit(out);
        hr = VariantCopy(out, &m_MyProperty);

        if(FAILED(hr))
                return hr;

        return S_OK;
}
```

Figure 55: Code demonstrating the copy of a `VT_BYREF`

performed. If the `VT_DISPATCH` object being pointed to is subsequently deleted, then you are left with a `VARIANT` pointing to an `IDispatch` object that no longer exists. If an attempt to get this property is subsequently made, the user will retrieve a `VARIANT` with a stale pointer, leading to the possibility of memory corruption.

5.3.1 The ActiveX Marshaller

In order to know the exact semantics of what happens to an object when it is passed as a parameter to an ActiveX control, you need to pay careful attention to what types the target function is expecting. When an ActiveX function expects a `VARIANT` as a parameter, the marshalling code does not do any sort of deep copy - it uses neither `VariantCopy()` nor `VariantCopyInd()`. So, receiving `VARIANT`s can be particularly dangerous if they contain COM interfaces that are operated upon beyond the method's scope. Furthermore, if an ActiveX function allows an indirect pointer to a COM object as a parameter - that is, (`VT_BYREF|VT_DISPATCH`) or equivalent, the object being referenced will have its reference count incremented by the marshaller (and released upon function returned). So if a `VARIANT` value is passed to an ActiveX control of type (`VT_BYREF|VT_DISPATCH`), it will not have its reference count incremented if the function takes a `VARIANT`, but it will have its reference count incremented if the function takes a `IDispatch **` (or even an `IDispatch *`). This algorithm is somewhat counterintuitive, which increases the likelihood that mistakes will occur as a result.

5.3.2 Detecting Shallow Copies with Static Analysis

Detecting shallow copies of COM objects is an opportunity for future research. This could be done with a data flow analysis similar to the ones described in Sections 5.1.4 and 6.1.3.1. To detect a shallow copy, one could develop a data flow algorithm that determines that a COM object is being passed to one of the many primitive memory copying operations (e.g., `memcpy()`). The interesting challenges to solve in that research would be similar to that which we did with C++. Performing the points-to analysis to determine that the source of the copy operation points to a COM object is a difficult task.

5.4 COM Trust Transitivity

The previous sections described type confusion vulnerabilities that can exist in C++ and COM as well as problems that arise when improperly copying COM objects. In this section, we desribe how the COM architecture can be used at runtime to subvert its own underlying security architecture.

5.4.1 Architectural Weakness

Once a COM object is installed on a system, it is available for instantiation by any application. There is no central policy governing which applications can load which COM objects. As vulnerabilities are disclosed in individual COM objects, there is nothing preventing an unknowing application from loading that potentially harmful control. Given that vulnerabilities in COM objects are most often left unpatched, any application that loads COM objects is a potential target an adversary. Specifically, an adversary can force an application to load one of the previously installed and known flawed controls allowing for exploitation. This allows adversaries to take advantage of controls that already exist on a system rather than attempting to trick their victim into installing an intentionally malicious object. Because many of the vulnerabilities that have gone unpatched allow for the execution of arbitrary code, an adversary can exploit these loitering flaws as an initial infection vector to install malware. Once they have installed their malware, they can gain complete control over the system. Given this exposure, many applications have implemented their own discrete policies dictating which controls they deem to be safe or unsafe. The Internet Explorer killbit list is an example of one such policy. These policies are only useful for determining which COM objects will be directly loaded by an application. They cannot guarantee or enforce the behavior of an object once it is instantiated. An application must therefore trust the behavior of a COM object once it is loaded. If that COM object then loads other COM objects, this trust is transitively extended. As shown in Figure 56, this trust transitivity exposes a critical security weakness with regard to COM. If an application loads a COM object that is within its security policy, it trusts that object to only load other objects that are also in the security policy.

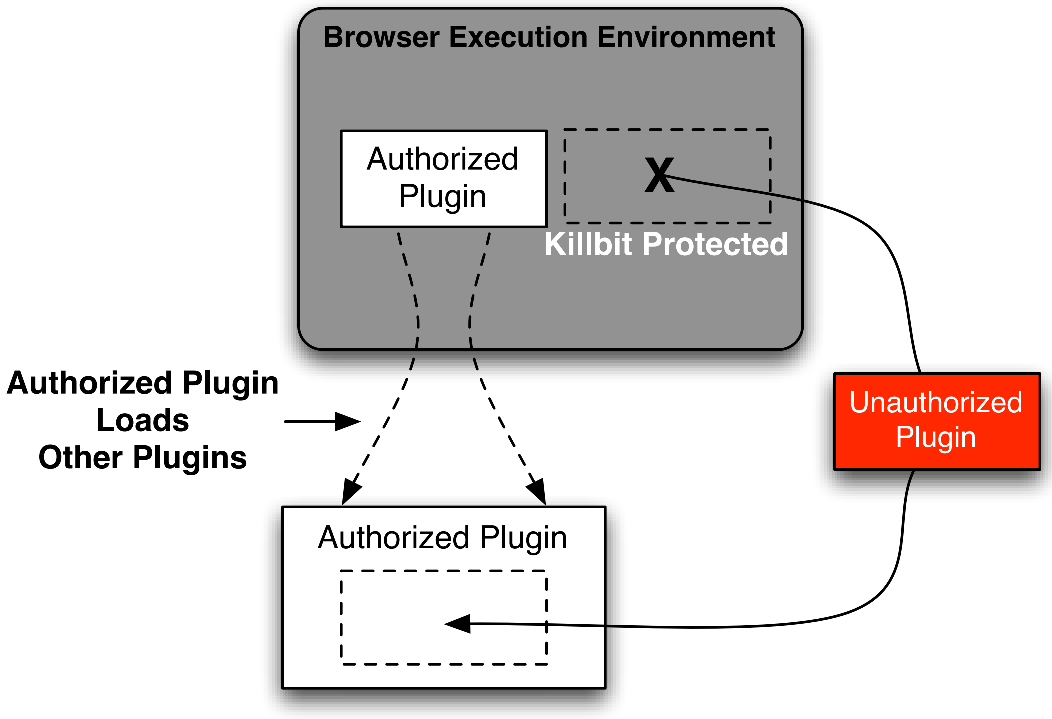


Figure 56: Transitive trust between COM Objects allows normally unauthorized objects to be loaded and executed.

There is no programmatic way for the application to ensure that an object it loads extends its security policy to other objects. It is therefore theoretically possible to exploit this transitive trust relationship in any application that loads COM objects. To complete an attack exploiting this transitive trust, an adversary requires the following:

1. An application that will render adversary-controlled content.
2. An application that will load COM objects.
3. A COM object that will in turn load other COM objects.
4. A vulnerable object that can be exploited.

Each of these requirements is easily achievable under normal circumstances with an average Windows installation. The following subsections address them in order.

5.4.1.1 Supplying Adversary-Controlled Content

The first requirement for accomplishing this sort of attack is that an adversary must persuade a target to render content under their control. This is easily accomplished through applications like Internet Explorer and the Microsoft Office suite. End users can be tricked into viewing malicious web pages through a number of means including phishing, cross-site scripting, and content injection. Additionally, users can be emailed Office and other documents or the browser can be used to force the rendering of Office content through the use of ActiveX.

5.4.2 Loading Adversary-Controlled COM Objects

COM objects can be loaded based on content in Internet Explorer, the Microsoft Office suite, as well as any other applications and services commonly installed on Windows that make use of this infrastructure including Flash and Adobe Reader. To force an application such as Internet Explorer to load a COM object, an adversary can supply the `CLSID` parameter of an `<OBJECT>` tag. In the case of the Microsoft Office suite, COM objects can be directly embedded in documents using the GUI.

5.4.2.1 Transitive Trust Amongst COM Objects

The third requirement for the attack is to load a COM object that will in turn load other objects. While this may seem rare, it is actually quite prevalent due to a feature of Microsoft Visual Studio. Because of the complexity of developing COM objects, Microsoft has included with Visual Studio a set of C++ templates called the Active Template Library (ATL). The ATL provides methods for the saving and loading of persistence streams so a developer does not have to understand the low-level details. To interact with these methods the developer simply provides a property map, a template defining the order and type of data that is stored in the persistence stream, for the object fields they intend to save stream. This way, as the control reads the persistence stream it understands to which properties it should apply the data. One weakness of property maps is the ability to define loose types. These are type

```
inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
{
        ATLASSERT(pStream != NULL);
        HRESULT hr;
        hr = VariantClear(this);
        if (FAILED(hr))
                return hr;
        VARTYPE vtRead;
        hr = pStream->Read(&vtRead, sizeof(VARTYPE), NULL);
        if (hr == S_FALSE)
                hr = E_FAIL;
        if (FAILED(hr))
                return hr;

        vt = vtRead;
        int cbRead = 0;
        switch (vtRead)
        {
        case VT_UNKNOWN:
        case VT_DISPATCH:
                {
                        punkVal = NULL;
                        hr = OleLoadFromStream(pStream,
                        (vtRead == VT_UNKNOWN) ? IID_IUnknown : IID_IDispatch,
                        (void**)&punkVal);
                        if (hr == REGDB_E_CLASSNOTREG)
                                hr = S_OK;
                        return S_OK;
                }
```

Figure 57: Source code for `CComVariant::ReadFromStream()`

```
HRESULT OleLoadFromStream(LPSTREAM pStm,
        const IID *const iidInterface, LPVOID *ppvObj) {

        CLSID pclsid;
        IID *riid;
        HANDLE *ppvObj;
        HANDLE *ppvStmObj;
        HRESULT hr;

        riid = iidInterface;

        if (*ppvObj == 0)
                return E_INVALIDARG;

        . . .

        if (!isValidInterface(pStm))
                return E_INVALIDARG;

        hr = ReadClassStm(pStm, &pclsid);
        if (hr)
                return hr;

        hr = CoCreateInstance(pclsid, NULL, CLSCTX_NO_CODE_DOWNLOAD|
                CLSCTX_REMOVE_SERVER|CLSCTX_LOCAL_SERVER|CLSCTX_INPROC_SERVER,
                riid, ppvObj);
        if (hr)
                return hr;

        hr = ppvObj->QueryInterface(IID_IPersistStream, ppvStmObj);
        if (hr) {
                ppvObj->Release();
                return hr;
        }

        hr = ppvStmObj->Load(pStm);

        ppvObj->Release();

        if (hr) {
                ppvObj->Release();
                return hr;
        }

        . . .

        return hr;
}
```

Figure 58: Approximation of `OleLoadFromStream()`

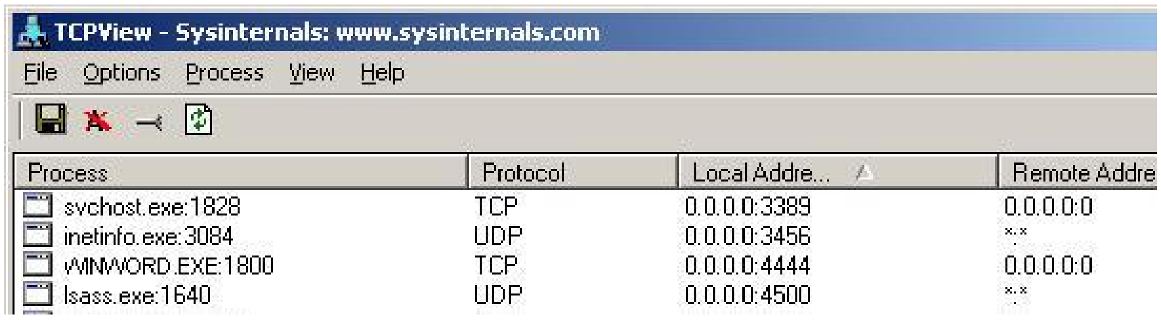
specifiers that do not directly map to one of the defined types. In other words, the property map can define that data exists in the persistence stream, but the COM object should examine the data itself to determine its type. This is the functional equivalent of a void pointer in the C programming language. By using the ATL, a COM developer can now easily create a COM object which implements persistence without understanding the underlying required methods or the COM-specific data types with which they are dealing. The critical issue then becomes the handling of loose-typed variants by the ATL-provided methods for loading a persistence stream. To read a persistence stream, the `IPersistStream::Load()` method provided in the ATL will call the method `CComVariant::ReadFromStream()`. As can be seen in the source code for `CComVariant::ReadFromStream()` in 57, there exist two variant types, that if encountered in the stream, will be passed to `OleLoadFromStream()`. An approximation of the source of `OleLoadFromStream()` obtained through reverse engineering the binaries is provided in 58. Here we can see that a CLSID is read from the data stream and passed to `CoCreateInstance()`. The remainder of the data stream is then passed to the `IPersistStream::Load()` method exposed by the object that has just been loaded. It is important to note that at no time was a security policy consulted before calling `OleLoadFromStream()`. This code distributed with the ATL can clearly be used to bypass the security policy of the parent application.

5.4.2.2 Finding a Known Flawed Control

The final requirement for this attack is to be able to load a vulnerable control. Given that vulnerabilities in COM objects are generally left unpatched, several hundred known flawed controls are resident on the average Windows installation. For example, on the system used to write this dissertation, the killbit list for Internet Explorer is over 600 entries in size. Each of those entries corresponds to known flawed control that is likely still unpatched.

5.4.3 Proof of Concept Attack

To demonstrate the severity of this attack, and the ease with which it can be accomplished on an average Windows platform, a working example was created leveraging well-known applications and commonly installed COM objects. The attack created for this proof of


TCPView - Sysinternals: www.sysinternals.com

File Options Process View Help

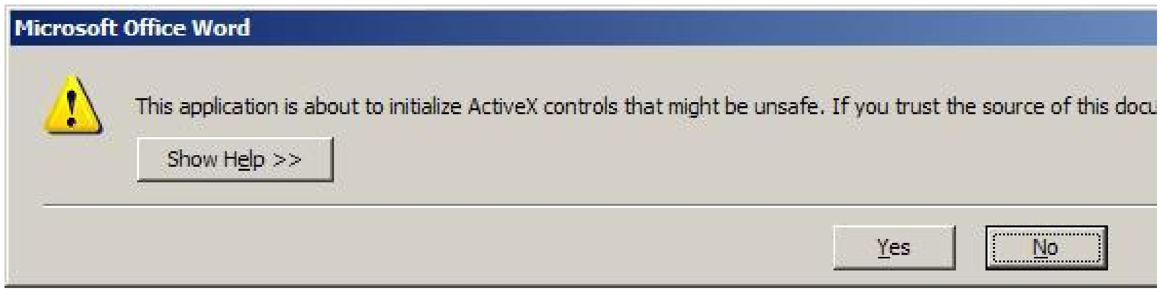
Process	Protocol	Local Addre...	Remote Addre
svchost.exe:1828	TCP	0.0.0.0:3389	0.0.0.0:0
inetinfo.exe:3084	UDP	0.0.0.0:3456	*:*
WINWORD.EXE:1800	TCP	0.0.0.0:4444	0.0.0.0:0
lsass.exe:1640	UDP	0.0.0.0:4500	*:*

Figure 59: The successful exploitation of a COM Object in Microsoft Word, demonstrated by having Word open a socket on port 4444.

concept loads a trusted COM object in Microsoft Word, which then loads a known flawed control that is still resident on most installations of Windows. The specific vulnerability used in this example is a known flawed control in all Windows XP installations and results in the execution of arbitrary code when triggered. We note that we tested different vulnerable controls on systems running Windows Vista and Windows 7 and were similarly able to compromise those systems. In this example, the COM object executes shellcode causing Word to listen on a TCP socket. Upon connection to this socket, the adversary is presented with a command prompt. While this section only describes an attack against Microsoft Word, the same attack was proven to be successful against several other COM containers including Microsoft WordPad, Microsoft Excel, Microsoft Powerpoint and Adobe Reader. To explain how the proof of concept attack was created, the following describes how each of the requirements in Section 5.4.1 was met.

1. Microsoft Word was chosen as the parent application for our proof of concept attack. Microsoft Word documents can be easily emailed to users. Additionally, the browser can be used as an intermediary in this attack by providing a link to a .doc file or using ActiveX to force Word to open a document of the adversary's choosing.

2. Microsoft Word was chosen for the ease with which COM objects can be embedded in documents. Object insertion is something that is done regularly in the course of authoring a document. This typically comes in the form of inserting images, tables, etc. In many cases these operations are actually embedding COM objects. This same process can be used to insert objects of many different types as long as they conform



This application is about to initialize ActiveX controls that might be unsafe. If you trust the source of this docu

Figure 60: Pop-up Warning from Microsoft Word When Attempting to Instantiate an Out-of-Policy Control

to the Microsoft Word security policy.

3. The Microsoft Date and Time Picker control was chosen because it provides the functionality to load subsequent COM objects by providing a CLSID in a persistence stream.

4. The final requirement for constructing the attack is to have the COM object loaded in the previous step then load a known vulnerable control. We selected Microsoft's Helper Object for Java, which contains a long standing, unpatched, exploitable vulnerability reachable by instantiating the control outside the Microsoft Java Virtual Machine.

To summarize, we create a working exploit by crafting a Microsoft Word document containing the embedded benign Microsoft Date and Time Picker control. That COM object in turn loads a known flawed control, thus circumventing Word's COM security policy. Once the flawed control is loaded, it triggers the vulnerability, resulting in the execution of arbitrary code. This attack, if executed in the real world, would easily enable an adversary to take full control of a victim's workstation to install malware or use the system for other arbitrarily malicious purposes. The known vulnerable COM object loaded in this attack is listed in the Internet Explorer killbit list. As such, it is the policy of Microsoft Word to not load this control. Had the attack simply tried to embed the control directly in the Word document, the security policy would have been effective, and the user would have been presented with the dialog shown in Figure 60. While the effectiveness of such warning messages is debatable, our attack allows for the vulnerable control to be loaded without providing the user with any indication that something is amiss. As shown in Figure 59, by using a trusted COM object to load the known flawed control, the security policy is bypassed, and the winword.exe process is now listening on TCP port 4444. This is demonstrable evidence that Microsoft Word has been compromised. In our specific exploit, upon connecting to the socket, the adversary is presented with a command prompt. This allows for the execution of any command in the security context of the user viewing the Word document. In a real attack scenario, an adversary would generally then go on to install their malware – taking complete control of the system.

5.4.4 Breadth of Attack

The attack described above is not unique to Microsoft Word, the Microsoft Date and Time Picker, and the Helper Object for Java. On the average Windows-based system, there are a large number of applications that would meet the first and second criteria for exploitation. In our testing, identified dozens of COM objects that exist on an average system that were either compiled with the ATL or expose functionally equivalent capability. As previously mentioned, the Windows XP-based system used to write this dissertation has several hundred known flawed controls still installed. This system is not unique. Given these numbers, the permutations of application, trusted COM object, and untrusted COM object quickly become unmanageably large. It is therefore unrealistic to require each application to attempt to main-tain security policies that can reasonably deal with this threat. It is clear that Windows requires an operating system-level security policy governing the instantiation of COM objects.

5.4.5 Preventing COM Trust Transitvity With Runtime Enforcement

In this section, we describe a solution we developed known as COMBlocker. This is a runtime enforcement system that provides a comprehensive view of COM instantiation allowing for complete mediation as compared to the relatively limited killbit methodology.

5.4.5.1 Mitigation Architecture

The lack of a central security policy governing the instantiation of COM objects has been identified as a major source of vulnerability in this work. In this section, a prototype of a mechanism which we call COMBlocker is proposed that introduces an operating system-level security policy with reference monitor-like functionality. This system will be used to enforce a security policy on the instantiation of all COM objects.

Design Goals and Assumptions The goal of COMBlocker is to provide a system-level policy for the instantiation of all COM objects. If every instantiation is monitored by a central policy, the issue of transitive trust can be remedied. The design of COMBlocker assumes that it is attempting to prevent the initial infection vector described in the previous section. It is not designed to secure a previously infected machine. Additionally, the prevention mechanism is designed to secure the instantiation of COM objects by applications that conform to Microsoft's design model. This is not intended to prevent intentionally malicious applications from loading flawed controls. These assumptions follow a common theme: the goal is to prevent the initial attack.

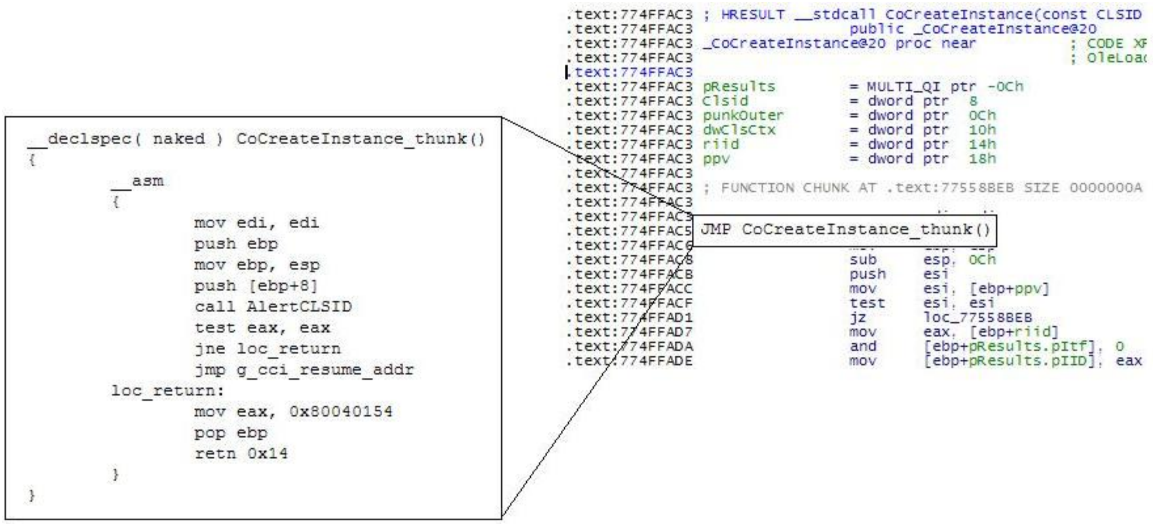
High Level Architecture COMBlocker is designed to interpose itself in the instantiation of all COM objects. In terms of the attack described in Section 5.4.1, the COM object loaded by Microsoft Word would be matched against a security policy and when that COM object in turn attempts to load another COM object, that subsequent instantiation would also be checked against the policy. To create such a policy enforcement system, the instantiation logic must be injected into every process on the system. This can be accomplished by a number of methods with varying levels of complexity and completeness. For reasons detailed later, COMBlocker injects a dynamically-linked library into every running process. This library contains the logic required to enforce the security policy. Once the COMBlocker library is injected into every process, calls to the COM instantiation APIs need to be redirected to the library to verify any object being loaded. We accomplish this through binary patching in our prototype. As the COMBlocker library is loaded, it locates the four COM instantiation APIs and overwrites the function prolog with a jump to the policy verification code. With the binary hooks in place, any application that calls any of the COM instantiation APIs is redirected to the security interface. This redirection will take place regardless of the source of the call to the instantiation API. In other words, the call to load a COM object could come from a base executable, a library, a different COM object, or any other type of executable code. Any call to any of the instantiation APIs is verified against our security policy. Once we can verify the instantiation of an object, a suitable policy must be defined. For the sake of simplicity our proof of concept starts by allowing the user to apply the Internet Explorer killbit list to all COM object instantiations. From there, a user can create exceptions to the killbit list or add disallowed objects on a per-application or system-wide basis.

5.4.5.2 Detailed Architecture

COMBlocker was developed as a third-party add-on for Microsoft Windows. It is not implemented by changing the underlying COM architecture or by invoking any extended APIs. If Microsoft were to implement a similar system, they could simply change the COM instantiation APIs to always check a central policy. The proof of concept developed in this research is a prototype of a system that a third-party could implement to provide a central COM security policy. With that, the details of how the solution was implemented are covered for completeness.

DLL Injection To introduce the security policy verification code into every running process, COMBlocker uses DLL injection. This is the process by which an application forces another application to load a DLL of its choosing. There are several ways to inject a DLL into another process, but since our solution requires injection into every process, we chose to use the `AppInit_DLLs` registry key [56]. The operating system will load all DLL's specified in this key into every process, thereby providing comprehensive coverage for all applications.

Binary Hooking Once the library is injected into every process, control flow from the COM instantiation APIs must be redirected to the COMBlocker security policy. Our proof of concept accomplishes this redirection through binary patching. The binary patching used by COMBlocker is very similar to the Detours API created by Microsoft Research [40]. In the case of COMBlocker, the function prolog of every COM instantiation API is overwritten with code that will jump to the security policy verification code in our library. As part of



```
__declspec( naked ) CoCreateInstance_thunk()
{
        __asm
        {
                mov edi, edi
                push ebp
                mov ebp, esp
                push [ebp+8]
                call AlertCLSID
                test eax, eax
                jne loc_return
                jmp g_cci_resume_addr
        loc_return:
                mov eax, 0x80040154
                pop ebp
                retn 0x14
        }
}
```

```
.text:774FFAC3 ; HRESULT __stdcall CoCreateInstance(const CLSID
.text:774FFAC3                 public _CoCreateInstance@20
.text:774FFAC3 _CoCreateInstance@20 proc near          ; CODE XF
.text:774FFAC3                                         ; OleLoac
.text:774FFAC3
.text:774FFAC3 pResults        = MULTI_QI ptr -0Ch
.text:774FFAC3 Clsid           = dword ptr  8
.text:774FFAC3 punkOuter       = dword ptr  0Ch
.text:774FFAC3 dwClsCtx        = dword ptr  10h
.text:774FFAC3 riid            = dword ptr  14h
.text:774FFAC3 ppv             = dword ptr  18h
.text:774FFAC3
.text:774FFAC3 ; FUNCTION CHUNK AT .text:77558BEB SIZE 0000000A
.text:774FFAC3
.text:774FFAC3
.text:774FFAC5 JMP CoCreateInstance_thunk()
.text:774FFAC6
.text:774FFAC8                 sub     esp, 0Ch
.text:774FFACB                 push    esi
.text:774FFACC                 mov     esi, [ebp+ppv]
.text:774FFACF                 test    esi, esi
.text:774FFAD1                 jz      loc_77558BEB
.text:774FFAD7                 mov     eax, [ebp+riid]
.text:774FFADA                 and     [ebp+pResults.pItf], 0
.text:774FFADE                 mov     [ebp+pResults.pIID], eax
```

Figure 61: Hooking Architecture for COMBlocker. Using binary rewriting, we force all COM objects to be checked against the global policy at their instantiation. Note on the left the policy check – `call AlertCLSID; test eax, eax;`

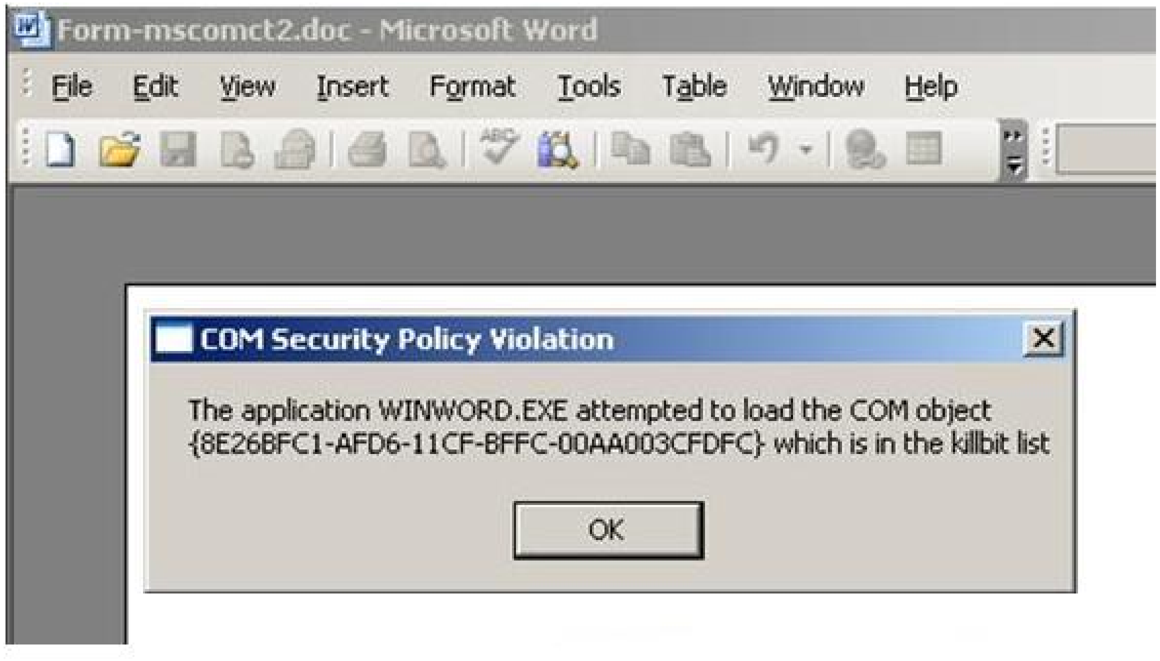
the DLLMain() function, our library will locate the four instantiation APIs and overwrite the first five bytes of the function prolog with a jump to the code that implements the policy verification. Since the first five bytes of the function have been overwritten, the first few instructions of the policy verification code must reproduce the functionality of the original API. Once these instructions are executed, the security policy enforcement can commence. Figure 61 shows the interception of control flow by COMBlocker to check against a global policy.

Enforcement Logic Once the control flow has been redirected from the instantiation APIs to our own logic, we then have access to the arguments to those APIs. This provides us with the necessary information to create an enforcement policy. Specifically, the CLSID of the object being instantiated is passed to the instantiation APIs as the first argument. In our enforcement logic, we retrieve a pointer to the CLSID from the stack of the instantiation API. If the CLSID is specifically blocked by the security policy in the registry, the enforcement logic simply returns the error `REGDB_E_CLASSNOTREG` to the calling application. This returns an invalid handle that the calling application cannot use for interacting with the object. All applications tested gracefully handled this error condition, but if one did not, it would crash rather than allowing the exploit to continue.

Policy Definition In COMBlocker, we wished to have a simple starting point to define the security policy. As such, the Internet Explorer killbit list can be applied to any application on the system or to the system as a whole. This is accomplished by setting a value in the registry in a location created by COMBlocker. The enforcement logic checks to see if this value is set. If it is, the killbit list is read from its default location in the registry and used for comparison against the CLSID the application is attempting to load. It is important to note that this is different than the security policy currently employed by Microsoft Word. In Word, the policy is only applied to the objects instantiated by the base executable. In our system, all instantiations are monitored. Once the killbit list is applied to an application, exemptions or additions to the list can be manually entered. These modifications are also set in the COMBlocker registry hive. For each application, subkeys in the registry are used to define by CLSID which objects are specifi- cally allowed or specifically denied. This allows the user to create detailed white lists and black lists for every application or the system as a whole. As mentioned in Section 5.4.4, it is unrealistic for a user to try to maintain lists of controls that should be considered secure and insecure. As such, it is the vision for a system like this to be implemented by Microsoft or a third party vendor. In these cases, Microsoft (or the vendor) could keep track of the known flawed controls and ensure they cannot be instantiated by any application except those specifically requiring them. Additionally, applications could be profiled to enumerate only those controls that should be instantiated under normal operation. In cases where this is possible, detailed white lists could also be created.

5.4.5.3 Results and Discussion

Breadth of Vulnerability The first objective described in this dissertation was to determine the breadth of the vulnerability. The issue of transitive trust amongst COM objects was hypothesized to exist in all COM containers which load content provided by a third party. This proved to be true for every application that was tested. Throughout the research, the attack described in Section 5.4.1 was reproduced in Microsoft WordPad, Microsoft Excel,


Form-mscomct2.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
COM Security Policy Violation
OK

The application WINWORD.EXE attempted to load the COM object {8E26BFC1-AFD6-11CF-BFFC-00AA003CFDFC} which is in the killbit list

Figure 62: COMBlocker Successfully Stopping Instantiation

Microsoft Powerpoint and Adobe Reader. The question of how additional third party COM containers might behave was also approximated. Microsoft Visual Studio 6 ships with a utility called the ActiveX Control Test Container. This utility allows developers to test the functionality of COM objects without having to create their own COM container. This loosely represents how a generic COM container would behave under normal circumstances. The attack from Section 5.4.1 was also successful in this tool, meaning that a significant number of other third-party applications are also vulnerable to these attacks. Each application tested was coerced into loading a COM object that was in direct violation of its security policy (where one exists). Generally speaking, these security policies are used to prevent the instantiation of COM objects that are known to contain vulnerabilities. In many cases, these security policies are used in lieu of fixing the vulnerabilities. With the research presented in this work, each of the loitering vulnerabilities in those controls can be resurrected and used for successful compromise.

Effectiveness of the Solution We tested COMBlocker's effectiveness and measured the overhead it imposes on a standard desktop system. The first step in testing the effectiveness of the solution was to apply it to each of the applications that were found to be vulnerable in the section above. For each application that was shown to be vulnerable, COMBlocker presented the user with the dialog box shown in Figure 62. This dialog box shows that the application attempted to load a control that is specifically denied by the defined policy. It also indicates that the instantiation of the object was prevented. COMBlocker was successfully able to prevent the attack described in Section 5.4.1 in Microsoft WordPad, Word, Excel, and PowerPoint as well as the ActiveX Control Test Container. Demonstrating the formal completeness of our solution is difficult. Our mechanism is helped by the fact that there are only a small number of publicly knownmeans by which COM objects can be instantiated. Injecting COMBlocker at these points should logically prevent applications from circumventing policy enforcement. However, if applications can instantiate COM objects through other unknown means such as implementing their own APIs, these interfaces would also need to be modified and mediated.

Performance A version of COMBlocker was created that logged the time required for each policy lookup encountered throughout the operation of an application. The test build was installed on a typical development workstation and gathered information for all of the COM object instantiations that occurred during a single day as part of a developer's normal work. The test workstation was a Windows XP SP3 machine with Office 2007, Internet Explorer 7, Firefox 3, Lotus Notes 8, Visual Studio 6, and several other commonly installed applications. During the course of the day, the behavior of the developer caused over 65,000 COM instantiations. Each of these recorded an average policy lookup time of 554µs to complete, with a 95% confidence interval of ±104µs. The variation in lookup time is largely due to the fact that consulting the killbit list in the registry is accomplished through a linear scan of the subkeys; it is not indexed. Testing shows that an average application incurs less than 10 policy lookups per user action. With that, each user action generates less than 5ms of delay due to COMBlocker. Another data point gathered in this test is that in general, Office applications and web browsers incurred a lower lookup time than core operating system components. When the data set is reduced to only Of- fice applications and web browsers, the average lookup time drops to 104µs, with a 95% confidence interval of ±14.2µs. This indicates that if performance were an issue in implementing a system like COMBlocker, the scope of the protection could be reduced to only those applications that are more easily targeted in COM-based attacks.

Discussion on Policy Creation As mentioned in Section 5.4, the base policy for each application (or the system as a whole) was the Internet Explorer killbit list. It appears that over time, several COM objects required for the normal operation of many of the applications tested have been killbitted. In other words, there exist several COM objects that are critical to the operation of Microsoft Word, Excel, and even Internet Explorer that are in the killbit list. The question arises: How can COM objects critical to Internet Explorer end up in the killbit list? The answer is that the killbit list blocks the instantiation of COM objects by the IE scripting engines, not the base IE executable. An example of this occurred when COMBlocker applied the killbit list to Internet Explorer as a whole, the navigation bar was prevented from being instantiated. The takeaway from this result of the testing is that the killbit list cannot be blindly applied to every application or the system as a whole. As policies are created, the killbit list can be used as a base, but modifications are required for each application being monitored. Any entity choosing to provide a solution like COMBlocker must take great care to ensure the security policies they define will allow for the normal operation of each monitored application while still providing a suitable level of security.

Future Enhancements to COMBlocker As mentioned above, policy creation can be quite complex. While the killbit list is a good starting point for a blacklist, it simply does not apply as-is to every application on the system. A useful enhancement to COMBlocker (or an accompanying tool) would be one that allows for the profiling of applications under normal use. This would help to expedite the identification of controls listed in the killbit list that are critical to the operation of other applications. Additionally, more research is required to determine the feasibility of runtime analysis of persistence streams. It could be possible to analyze data contained in a persistence stream and filter access to that data based on policy. For example, one policy could be to ensure that COM objects may only load simple data types and cannot load the more complex types, which could force the instantiation of other objects.

Comparsion to Microsoft-Issued Patch On June 8, 2010, Microsoft released Security Bulletin MS10-036 in response to our private disclosure of this vulnerability. This patch attempts to prevent the attacks discovered and demonstrated in this dissertation. Specifically, this patch extends the killbit list to nested COM instantiations made by Microsoft Office. While this patch effectively prevents Office-generated files from including these attacks, it does not protect any other application that takes advantage of the COM infrastructure. Accordingly, this patch does not provide the security guarantees of COMBlocker, and therefore means that Windows-based systems still remain vulnerable to such attacks through other applications.

Chapter VI

DELETION PROBLEMS

In the previous two chapters, we discussed the sorts of security issues that can arise with complex object types during the instantiation and lifetime of the object. In this chapter, we discuss the sorts of issues that can arise as the objects are deleted.

6.1 C++ Use-After-Free Conditions

In this chapter, we present a static analysis technique to detect use-after-free conditions in compiled binaries. Because the structure of how C++ objects are compiled and stored in memory is crucial to both the understanding of use-after-free vulnerabilities as well as analysis of C++ binaries, we give a brief description of how C++ objects are represented after compilation. We then give a short description of how use-after-free vulnerabilities occur in code and how they are exploited. We conclude this background section with a review of available expression analysis from compiler theory.

6.1.1 C++ Objects in Memory

One of the tasks of a C++ compiler is to ensure the correct storage of C++ classes in memory while providing both multiple inheritance and virtual functions. C++ compilers store instantiated objects as contiguous structures in memory. This is fortunate because it simplifies the discovery of object instantiations in compiled binaries. Our discussion here is focused on how Visual Studio compiles C++; other compilers represent C++ objects in a similar manner. Classes without virtual functions are represented in memory like traditional C structs. Simply, the properties (data members) of the object are placed in contiguous memory in order of declaration in the code. The object's method calls are compiled as direct calls to functions that take a pointer to the object as an argument (the this pointer). When an object has one or more virtual functions, virtual function calls must be dispatched at runtime through the use of a virtual function table (vtable). A vtable for an object is a contiguous set of function pointers that point to the appropriate functions for that object type. The virtual function can be called by finding the correct function pointer in the function pointer table. Objects with virtual methods are laid out in memory the same way as objects without virtual methods, but have an initial member element: a pointer to the class's vtable. This is a pointer to an array of function pointers (one for each virtual funciton) stored in the read-only .data section of the binary. When a class inherits from a single base class, the class instance begins with a vtable pointer, followed by the base object properties and then the derived object properties. When a class inherits from more than one class, the base class layouts are placed consecutively, with the derived class's data members following. Classes that are the result of multiple inheritance have more than one vtable—one for each base class. Virtual methods from the derived class will be appended to the vtable for the first base class. The memory structures described above are created on the stack or the heap by the class constructor depending on how the object is allocated in the source code. The compiler may also choose to make the constructors inline or create a sepa-

Note that the presence of a command line argument at line 18 causes object a to be deleted, even though it is called again at line 26. rate function call for the constructor. In Visual Studio, this is a configurable optimization that can be set by the developer.

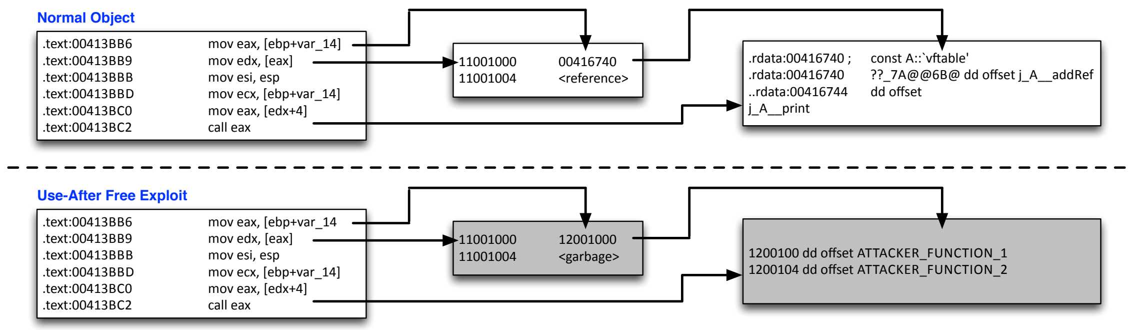
6.1.2 Use-After-Free Vulnerabilities

Use-after-free (UAF) vulnerabilities are a class of software flaws that involve using a memory resident object after it has been freed. While, use-after-free conditions are possible and prevalent in C, they are reported as security vulnerabilities more frequently in C++. UAF vulnerabilities most commonly occur when a C++ object that was allocated on the heap is accessed after it is deleted, but stack-allocated objects can also be used after a free. Developers can easily make this memory management error, especially in large and complex codebases, and often in an attempt to prevent memory leaks. This creates a condition where an object is deleted on some code paths, but not all. The code in Figure 63 provides a simplified example. In this example, the developer chose to delete the object if the first

```
class A {
private:
        int reference;

public:
        A();
        ~A() { };
        virtual void addRef();
        virtual void print();
};

int main(int argc, char* argv[])
{
        A *a = new A;

        a->addRef();
        a->print();

        if (atoi(argv[1]) == 1) {
                delete a;
        }

        //. . .
        //Memory allocation operations
        //. . .

        a->print();

        return 0;
}
```

Figure 63: Sample C++ code with a use-after-free vulnerability.

command line argument is “1”, but needed the object under all other conditions. If an attacker can cause the program to take the first branch, the object is deleted, but then is used later in the program. When an object is deleted, new data takes the place of the previously deleted object’s properties and/or vtable pointer. Later, if the deleted object’s property is accessed, this new data could be read or modified, affecting the reliable operation of the program. If a deleted object’s virtual method is called, whatever data has been written to the old vtable pointer will be dereferenced to locate the correct function pointer. This can result in memory access violations or otherwise unstable execution. In addition to the hazards of a normally functioning program with a use-after-free condition, it can also be a path to software exploitation. While a use-after-free can result in an attacker with the ability to write data to the stack or heap to influence decisions made based on an object’s properties, the greatest danger is if the use-after-free includes a virtual function call. In that case, the attacker can write his own vtable containing pointers to shellcode and fill the deleted object’s memory with references to that vtable. On the virtual function call, execution will transfer to the attacker’s injected code. Figure 64 demonstrates a use-after-free exploit of the code shown in Figure 63. The left side of Figure 64 shows a fragment of the compiled code from Figure 63. That fragment completes with a call to the virtual function `print()` at line 413BC2. The object pointer is consulted to find the pointer to the vtable at line 413BB9. At runtime, the object is stored on the heap — in this example, the object is stored at address 11001000, and the class’s vtable is located at address 416740. The function pointer stored at offset 4 represents the second virtual function declared in the class (since function pointers are 4 bytes each). If this object were overwritten using a use-after-free vulnerability, and the attacker could control the value of the data at the heap location where the object was stored (which in many cases is not difficult), the attacker could overwrite the vtable pointer with a pointer to the attacker’s own vtable. The bottom half of Figure 64 shows this scenario with gray boxes. In that example, the attacker has established a vtable pointer of 12001000 (also heap data written by the attacker) and has preloaded that location with function pointers to malicious functions (again, written by the attacker). When the second function in the vtable is called, the processor will call address `dd offset`



```
.text:00413BB6        mov eax, [ebp+var_14]
.text:00413BB9        mov edx, [eax]
.text:00413BBB        mov esi, esp
.text:00413BBD        mov ecx, [ebp+var_14]
.text:00413BC0        mov eax, [edx+4]
.text:00413BC2        call eax

11001000    00416740
11001004    <reference>

.rdata:00416740 ;    const A::`vftable'
.rdata:00416740      ??_7A@@6B@ dd offset j_A__addRef
..rdata:00416744     dd offset
j_A__print
```

```
.text:00413BB6        mov eax, [ebp+var_14
.text:00413BB9        mov edx, [eax]
.text:00413BBB        mov esi, esp
.text:00413BBD        mov ecx, [ebp+var_14]
.text:00413BC0        mov eax, [edx+4]
.text:00413BC2        call eax

11001000    12001000
11001004    <garbage>

1200100 dd offset ATTACKER_FUNCTION_1
1200104 dd offset ATTACKER_FUNCTION_2
```

Figure 64: Memory layout of a valid object and after a use-after-free exploit

`ATTACKER_FUNCTION_2` instead of address `dd offset j_A_print`. Unfortunately, compilers fail to detect use-after-free vulnerabilities. Not only do they miss complex flaws that are the result of interprocedural memory management, they also miss trivial examples like the one in Figure 63. Visual Studio 2012, g++ 4.8.3, and clang++ 3.5 all fail to detect the flaw in that exact code sample.

6.1.3 Detecting Use-After-Free Conditions With Static Analysis

6.1.3.1 Available Expression Analysis

Available expression analysis (abbreviated AVAIL) is a common compiler data flow analysis that enables common subexpression elimination. AVAIL takes as input a control flow graph consisting of basic blocks, and AVAIL identifies, for each basic block, a set of expressions that will always be computed before the entry of a given basic block. Knowing the set of expressions that are always computed before a basic block allows a compiler to set aside results to avoid redundant computation (specifically, common subexpressions). While AVAIL is usually presented as a local (within a single procedure) algorithm, interprocedural available expression analysis is also possible. More specifically, AVAIL computes four sets for each basic block B: $GEN[B]$, $KILL[B]$, $AVAIL_{IN}[B]$, and $AVAIL_{OUT}[B]$. $AVAIL_{IN}$ and $AVAIL_{OUT}$ refer to the expressions that are available before and after each block (respectively). GEN is the set of new expressions that are defined in the basic block, and $KILL$ is the set of expressions whose values have changed because a variable used in the expression has changed. These sets are computed with the following two relations:

$$AVAIL_{IN}[B] = \bigcap_{p\ proceeds\ B} AVAIL_{OUT}[p] \tag{3}$$

$$AVAIL_{OUT}[B] = GEN[B] \cup (AVAIL_{IN}[B] - KILL[b]) \tag{4}$$

where p is a basic block (that proceeds a block B). In our analysis, objects must be tracked interprocedurally. In those cases, $AVAIL_{IN}$ at the entry of a function F is equal to $AVAIL_{OUT}$ at the call site to F from a call site c. Within a basic block, we say that an expression is in the "$AVAIL$set" at an execution point a if it has been generated before a, or if the expression is in $AVAIL_{IN}$ and not killed before point a. These relations, when written for every basic block, form a system of equations. Because the control flow graph contains back-edges (from loops and other control structures), and thus blocks can be affected by both preceding and subsequent blocks, solving the system of equations requires a fixed-point algorithm (that runs through each block iteratively until the four sets for every block do not change). The recurrence relationship described above forms the basis of many other data flow analysis algorithms, with slightly different definitions of IN, OUT, GEN, $KILL$, and the socalled "meet" operator, which in the case of $AVAIL$ is set union. In this work, we define a new data-flow analysis algorithm in the following section. Our algorithm uses the same recurrences above, but with new definitions of GEN and $KILL$ that allow us to track object instantiation and deletion rather than availability of expressions.

6.1.3.2 *Available Object Definition Analysis*

As discussed in the previous section, use-after-free vulnerabilities occur when an object is accessed after being freed. Frequently, the object is freed along some code paths, but not others. Accordingly, use-after-free errors can be detected by finding every access where the object may have been deleted on some proceeding code path. Thus, an analysis that determines if all code paths to an access contain a valid object definition will detect use-after-free conditions. In this section, we define such an analysis and term it Available Object Definition Analysis (AODA). Our crucial insight is that the recurrence relations from $AVAIL$ can be used to track the availability of instantiated objects to detect use-after-free errors. To track instantiated objects, we simply need to redefine how the GEN and $KILL$ sets are populated during the analysis. We will use the same notation for these sets, but note their alternative meanings in AODA. In AODA, GEN is the set of objects that are instantiated in a basic block, while $KILL$ is the set of objects that are freed in a basic block. $AVAIL_{IN}$ is simply the set of objects that are instantiated (and thus, available) along all code paths before a basic block, while $AVAIL_{OUT}$ are the objects that are available after a basic block has executed. Tracking all instantiated class pointers is significantly more difficult than tracking expressions. The following section details how we reliably compute the GEN and $KILL$ sets. Given GEN and $KILL$, computing $AVAIL_{IN}$ and $AVAIL_{OUT}$ requires a straightforward application of iteration over the control flow graph. We rely on the Recall framework to compute the control flow graph, and identify object instantiations and deletions. Although in this work we evaluate AODA on compiled binaries, as a data flow algorithm it can also be implemented within a compiler or within a stand-alone static analysis tool.

6.1.3.3 Implementation

In this section, we describe how we extended the Recall binary analysis framework to implement AODA. This section includes the technical details involved in identifying object instantiation, usage, and deletion. In the following section, we use this implementation to detect use-after-free conditions in several test applications adapted from pubilcly available projects as well as 652 MicrosoftWindows libaries and a previously disclosed use-after-free vulnerability.

Assumptions This research focuses on the common and important problem of identifying software vulnerabilities in compiled code. Because we focus on commercially available, release-build code, we make no claims about the applicability of our algorithms or implementation to malware or other hardened, obfuscated code. We chose to focus on Windows binaries compiled to x86 by Microsoft Visual Studio for this work. We chose this platform because the majority of closed-source software (which requires binary analysis) is implemented for that platform. While our platform choice influences our implementation and choice of experimental targets, the techniques we present will be applicable to any x86 binary format and compiler choice. For the sake of generality, we assume that Runtime Type Information (RTTI) is unavailable to the analysis. If this information is available, it can only make object detection (and our subsequent analysis) more accurate.

Identifying object instantiation and deletion To compute GEN for AODA, we must identify every object instantiation, and to compute $KILL$ for AODA we must identify every object deletion. When C++ code is compiled, the clear type definitions in the source code become calls to relevant constructors that create the memory objects described in Section 6.1.1, and accordingly are non-trivial to identify. We rely on RECALL's ClassTracker to identify C++ object creation and deletion with high reliability. RECALL uses four simple, reliable heuristics to identify C++ objects that reflect all possible ways a C++ object can be instantiated at runtime: Stack-allocated object with in-line constructor Heap-allocated object with in-line constructor Stack-allocated object with called constructor Heap-allocated object with called constructor The heuristics for detecting heap objects rely on the compiler's use of a specific function for the new operator. In the case of Visual Studio, after compilation this function is referred to as `YAPAXI`. Once an object instantiation is located, RECALL discovers an object's type by tracing the size of the new vtable and set of funtion pointers as well as the number and size of the properties of the object. When a new object is encountered, the virtual address of the constructor of the object is used as an opaque identifier for that object type. We extend RECALL to also detect object deletion. Deletion detection also differs between stack and heap declared objects. Stack-alloocated objects are deleted by first calling the object's destructor, then calling a delete operator (named `YAXPAX` Visual Studio after compilation). The compiler automatically deletes stack-declared objects when they fall out of scope. Heap-allocated objects are deleted only when the developer makes an explicit call to delete. When this occurs, Visual Studio creates a helper function that calls the object's destructor and the delete function `YAXPAX`. The name of the helper function will include the string scalar deleting destructor or vector deleting destructor depending on whether the developer calls `delete` or `delete[]`, respectively. In the case of heap-allocated objects, to ensure correctness RECALL traverses into the helper function to ensure that the class destructor is called and that there is an explicit call to `YAXPAX`. RECALL detects deletion of both types of object. In particular, because heap object deletion is so distinctive, we are highly confident in our ability to detect deleted objects on the heap.

Complex Real-World Scenarios While the basic concept of how AODA is able to identify object instantiations and deletions is covered above, in practice, developers employ many variations on this concept that complicate the analysis. The details of some of the more complex, real-world scenarios we identified are covered in the following subsections.

Virtual Destructors A developer may choose to declare the destructor for a class as virtual. This is important if the developer encounters a scenario where they wish to delete an instance of a derived class through a pointer to a base class. This however, introduces complexity for AODA. The problem is that when performing the data flow analysis, there is no clear call to the destructor for the object. Rather, an indirect function call is inserted into the binary. The structure of the object must be fully recovered to gain an understanding of what that indirect function call is doing. We use the RECALL framework to reconstruct the vtable of the object. Then, any indirect function call encountered during the data flow analysis is reconciled to its actual function, and further analyzed to determine whether it is actually the destructor for the object. Of course, following every indirect function call during the analysis can cause the runtime of the analysis to grow exponentially. To combat this issue, AODA takes a configuration parameter to define how deeply it should follow the indirect calls to look for the destructor. In practice, we found that a depth of one is sufficient to find virtual destructors. To further limit the impact of following indirect function calls, this traversal only occurs for functions which employ the thiscall calling convention (i.e., one that passes the this pointer as an argument in the ecx register). This ensures that the indirect function call is, in fact, a call to a class method and not some other arbitrary function.

Factory Design Pattern Much like virtual destructors require AODA to traverse function calls to identify a destructor, the factory design pattern requires AODA to traverse function calls to identify the instantiation of new objects. The factory design pattern is a common object-oriented design construct in which the developer creates a method which will instantiate differing objects based on the arguments specified. When this design pattern is used, the constructor for the object is not directly encountered during the data flow analysis. Rather, AODA must follow function calls to determine whether the returned value is a class pointer. In the same way AODA takes an argument to specify the depth to which it should traverse indirect calls to find destructors, the depth of function calls to be traversed to find constructors is configurable. Again, in practice, a depth of one was found to be sufficient. However, it is more difficult to limit the functions that will be traversed to find instantiated objects. When the constructor depth is set to one, it will follow every function call encountered during the data flow analysis to determine whether a class pointer is returned.

Computing AODA and Finding Use-After- Free vulnerabilities The second pass made by RECALL over the intermediate representation performs the fixed point availability analysis algorithm described in Section 6.1.3.1. In this pass, it performs a forward analysis iterating over each basic block. As object instantiation points are identified, the virtual address of the constructor is added to the GEN set for the given basic block. As object deletion points are identified, objects are added to the $KILL$ set. The analysis identifies which particular object is deleted by following the use-def chain to locate its instantation. As RECALL iterates to the next basic block, the objects in all the $AVAIL_{OUT}$ sets of the block's predecessors are added to the $AVAIL_{IN}$ set for the current block. This process is repeated in a fixed-point algorithm until $AVAIL_{IN}$ and $AVAIL_{OUT}$ sets have been generated for each basic block. In the third pass over the code, RECALL identifies usage points of binary objects by using a forward analysis that iterates over each basic block in the IR. In each basic block, RECALL looks for indirect function calls — calls to function pointers that are dereferenced from a larger containing data structure (i.e., a vtable). When RECALL identifies this condition, it traverses the use-def chain of the function pointer backwards to identify the instantiation point of the containing structure. If the instantiation point can be traced back to an object instantiation detected in the first pass, RECALL then determines whether the virtual address of the call to that object's constructor is present in the $AVAIL$ set for the basic block at the program point where the usage occurs. If the call to the object's constructor is not present in the $AVAIL$ set for the basic block where the usage occurs, a potential use-after-free condition has been discovered.

6.1.3.4 Results

After implementing the Availabile Object Definition Analysis described in Section 6.1.3.1, we tested the framework on a series of simple examples to ensure the algorithm was operating correctly. The process used for testing and the results of the tests are detailed in Section 6.1.3.4. Then, use-after-free conditions were injected into several publicly available projects as detailed in Section 6.1.3.4. By scanning these projects, we ensure that our analysis is correct for real-world applications. To ensure the analysis would work on a real-world vulnerability, it was run against a library with a known use-afterfree vulnerability. Section 6.1.3.4 details how the results of the analysis were verified. Once we verified the results on a control set of binaries, the framework was run over a substantial subset of the .dll's in the system32 directory on a default install ofWindows 7. Section 6.1.3.4 details the number of use-after-free conditions that exist unpatched on one of the world's most popular operating systems.

Control Set Results The first step in verifying the Available Object Definition Analysis was to test it on a series of simple examples and manually verify the results. An example of one of the simple examples can be found in Figure 63. This example code was compiled and then run through RECALL, which generated an LLVM bitcode file. The control flow graph of the resulting bitcode can be seen in Figure 65 (some code is omitted). Here we can see the object a is instantiated in basic block 413b20 (denoted as "1" in Figure 65). We can then see that the object is deleted in basic block 413c0b (denoted as "2" in Figure 65). Then the object is used in basic block 413c2a (denoted as "3" in Figure 65). However, since there is a path to 413c2a whereby the object is deleted, we have a potential use-afterfree vulnerability.

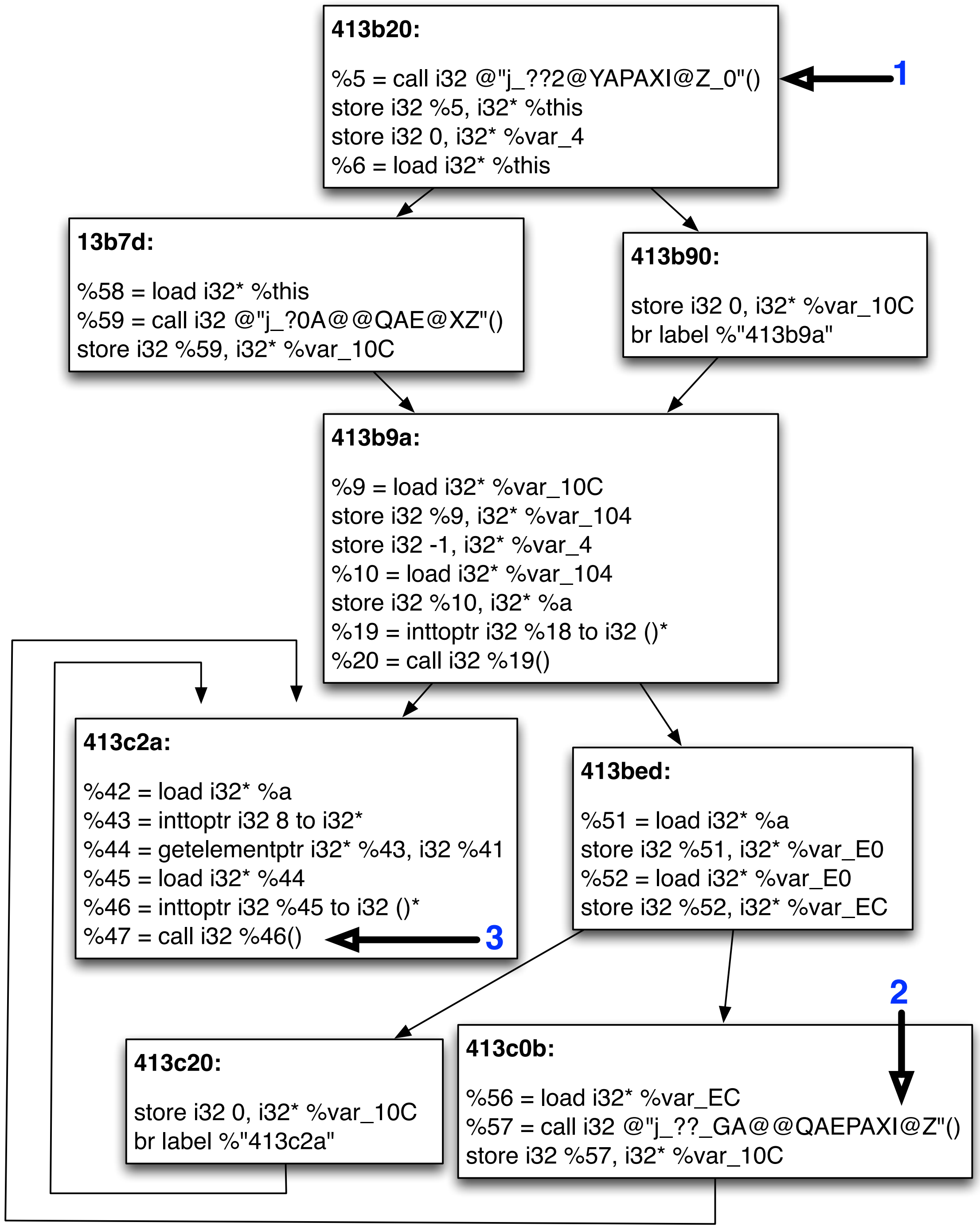

413b20:
%5 = call i32 @"j_??2@YAPAXI@Z_0"()
store i32 %5, i32* %this
store i32 0, i32* %var_4
%6 = load i32* %this
1
13b7d:
%58 = load i32* %this
%59 = call i32 @"j_?0A@@QAE@XZ"()
store i32 %59, i32* %var_10C
413b90:
store i32 0, i32* %var_10C
br label %"413b9a"
413b9a:
%9 = load i32* %var_10C
store i32 %9, i32* %var_104
store i32 -1, i32* %var_4
%10 = load i32* %var_104
store i32 %10, i32* %a
%19 = inttoptr i32 %18 to i32 ()*
%20 = call i32 %19()
413c2a:
%42 = load i32* %a
%43 = inttoptr i32 8 to i32*
%44 = getelementptr i32* %43, i32 %41
%45 = load i32* %44
%46 = inttoptr i32 %45 to i32 ()*
%47 = call i32 %46()
3
413bed:
%51 = load i32* %a
store i32 %51, i32* %var_E0
%52 = load i32* %var_E0
store i32 %52, i32* %var_EC
2
413c20:
store i32 0, i32* %var_10C
br label %"413c2a"
413c0b:
%56 = load i32* %var_EC
%57 = call i32 @"j_??_GA@@QAEPAXI@Z"()
store i32 %57, i32* %var_10C

Figure 65: Control flow graph of code from Figure 63. At point 1, the object a is declared. At point 2, the object is deleted. At point 3, `a->print()` is called again. Note that because point 3 is reachable via point 2, a use-after-free vulnerability is possible.

```
Processing Function: _main(i32, i*)
Processing Basic Block: 413b20
        Avail_in = {}
        Avail_out = {j_??0A@@QAE@X}   ## 1 ##
Processing Basic Block: 413b7d
        Avail_in = {j_??0A@@QAE@X}
        Avail_out = {j_??0A@@QAE@X}
Processing Basic Block: 413c2a
        Avail_in = {}                          ## 3 ##
        Avail_out = {}
Processing Basic Block: 413bed
        Avail_in = {j_??0A@@QAE@X}
        Avail_out = {j_??0A@@QAE@X}
Processing Basic Block: 413c0b
        Avail_in = {j_??0A@@QAE@X}    ## 2 ##
        Avail_out = {}
Processing Basic Block: return
        Avail_in = {}
        Avail_out = {}
Avail sets complete...

Processing Basic Block: 413b90
Processing Basic Block: 413b9a
        Found use of: j_??0A@@QAE@X
        In Avail Set...
        Found use of: j_??0A@@QAE@X
        In Avail Set...
Processing Basic Block: 413c2a
        Found use of: j_??0A@@QAE@X
        ERROR - Not in Avail Set...
        Potential Use After Free Condition
Processing Basic Block: 413bed
Processing Basic Block: return
```

Figure 66: AODA results show that compiled code of Figure 63 contains a use-after-free vulnerability

```
Avail sets complete...
Processing Basic Block: ed0c85e
Processing Basic Block: ed0c877
Processing Basic Block: ed0c881
Processing Basic Block: ed0c887
Processing Basic Block: ed0c883
Processing Basic Block: ed0c8aa
Processing Basic Block: ed0c8b6
Processing Basic Block: ed0c8f5
Processing Basic Block: ed0c8ff
Processing Basic Block: ed0c902
Processing Basic Block: ed0c8f9
        Found use of: ??0CEditSession@@QAE@P6GJKPAV0@@Z@Z
        ERROR - Not in Avail Set...
        Potential Use After Free Condition
```

Figure 67: AODA results detect use of an object that is not in the $AVAIL$ set in a previously-disclosed vulnerability

Table 5: Github projects with injected vulnerabilities

Project	Injected Vulns	Discovered Vulns
Leanify	1	1
libwebm	2	2
lifespan	1	1
MAPIEx	2	2
MonaServer	2	2
557-animator	2	2
BarsWF	1	1
easyrtc_ie_pluging	1	1
ElasticTabstopsForScintilla	1	1

When AODA is run on the control flow graph in Figure 65, it automates the detection logic described above. It does this in two passes over the code. The first pass identifies the $AVAIL$ sets as shown in Figure 66. The most notable points of the first pass are where the analysis detects the generation of a new object in basic block 413b20 (denoted as “1” in Figure 66), and where it detects an object being killed in basic block 413c0b (denoted as “2” in Figure 66). Then, since basic block 413c2a has three predecessors, and the required object is not in the $AVAIL_{OUT}$ sets for all of the preceding basic blocks, it is not added to the $AVAIL_{IN}$ set (denoted as “3” in Figure 66). The second pass over the code looks for uses of objects. It checks to make sure that the class pointer that is referenced is in fact in the $AVAIL_{IN}$ set of the containing basic block. In the case of the code in Figure 65, the statement `%47 = call i32 %46()` makes use of an object that is not contained in the $AVAIL_{IN}$ set (denoted as ERROR in Figure 66). Since, there exists a control flow path where the object is not available, a use-after-free condition exists at this program point. In addition to the code in Figure 63, we tested six other test cases involving use-after-free with various control structures. These tested if-else statements, switch statements, goto statements, and switches combined with if-else and goto, and a switch with an “accidental fall through.”

Manually Generated Vulnerabilities To further test the detection capabilities of Available Object Definition Analysis, it was tested on a series of publicly available projects.

To find suitable projects, we used a script to identify projects on Github that included Visual Studio project files and made use of heap-allocated objects. Of the first 1000 projects found on Github, 66 met both of those conditions. We selected the first nine projects that would build with minimal modification and injected vulnerabilities into the source. The vulnerabilities were injected using one of two methods. The first is by freeing objects that had been allocated by the original author. By using objects that were defined by the original author, and only injecting deletions of those objects, we encountered several real-world scenarios that were not seen in our control set tests described above. For example, the Leanify[1] project makes use of the factory design pattern and virtual destructors. As described in Section 6.1.3.3, these conditions make detection of use-after-free conditions more difficult. The second method for injecting use-after-free conditions is by deleting return statements that would cause the use of a freed object to be unreachable. After injecting the vulnerabilities into the nine test projects, they were each run through AODA. The results of those tests can be seen in Table **??**. In short, AODA was able to detect every one of the thirteen use-after-free conditions that were injected into the nine Github projects without a single instance of a false positive.

Real-World Vulnerability In addition to verifying AODA with simple examples, we also confirm that AODA can detect previously-disclosed vulnerabilities. In particular, we analyze a vulnerability in Microsoft Internet Explorer (located in tiptsf.dll) that was disclosed as MS13-069 and CVE-2013-3205. In Figure 67, we can see the $AVAIL$ sets as they were generated during analysis (some basic blocks omitted). Later we see a use of an object of type ??0CEditSession@@QAE@P6GJKPAV0@@Z@Z that is not in the $AVAIL_{IN}$ set of the basic block containing the use point. This indicates the presence of a use-after-free vulnerability. The preceding example represents an interesting case that was caught by the automated analysis. In this case, there was a use of a class pointer before it was ever initialized. While this is not strictly use-after-free vulnerability in the textbook sense[2], the

[1]https://github.com/JayXon/Leanify

[2]From the perspective of an adversary, this is exactly the same - a portion of improperly allocated memory is accessed by the program.

Table 6: DLLs containing use-after-free vulnerabilities

Library	Potential Use-after-free Conditions
ActionCenterCPL.dll	5
AdmTmpl.dll	16
bidispl.dll	3
cabview.dll	12
certcli.dll	1
cewmdm.dll	8
cnvfat.dll	1
comdlg32.dll	1
comsnap.dll	9
credui.dll	6
cscapi.dll	2
dataclen.dll	2
devenum.dll	3
devmgr.dll	2
eapp3hst.dll	2
eappcfg.dll	4
Faultrep.dll	2
FirewallAPI.dll	1
fontext.dll	5
fundisc.dll	4
gpprnext.dll	1
gpedit.dll	3
iasrad.dll	1
icsigd.dll	1
imagehlp.dll	1
ipsmsnap.dll	23
itss.dll	8

analysis was able to detect it. This is because there was a program point where class type was used, but not in the $AVAIL_{IN}$ set for the containing basic block. Since there is a code path in which the class pointer was not instantiated, it was not included in the $AVAIL_{IN}$ set.

System32 Directory Results To estimate the extent of use-after-free vulnerabilities in production operating systems, we ran AODA on 652 binaries found in the system32 directory on a default install of Windows 7. The analysis revealed 127 potential use-after-free conditions. Table **??** shows the results of the scan of the binaries. These libraries are often used by popular applications (and are thus popular targets for attackers) like Microsoft

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
        if(pCallback == NULL)
                return E_INVALIDARG;

        pCallback->AddRef();
        m_pCallback = pCallback;

        return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
        if(out == NULL || *out == NULL || m_pCallback == NULL)
                return E_INVALIDARG;

        *out = m_pCallback;

        return S_OK;
}
```

Figure 68: Sample code showing a reference counting overflow

Office and Internet Explorer.

6.2 COM Release Failure

Failure to release an object essentially amounts to a memory leak. These failures occur when a COM interface is referenced through `IUnknown::AddRef()` or `IUnknown::QueryInterface()`, and are later discarded without calling the corresponding `IUnknown::Release()` function. Triggering code paths that operate this way can allow an attacker to consume arbitrary amounts of memory, but more usefully increment the reference count of an object an unlimited number of times. On a 32-bit machine, by executing the vulnerable code path 0xFFFFFFFF times, an integer overflow can be triggered in the object's reference count. Following that, any call to `IUnknown::Release()` will cause the object to be de-allocated, which, again, can lead to stale pointer problems. The code in Figure 68 is based on an example we previously used; however, it has been modified to demonstrate problems with failing to release an object.

This example correctly adds a reference to the new callback object when it is set. However, the previous value held in `m_pCallback`, if one existed, is overwritten without being released. Therefore, an attacker can set this property a large number of times and eventually

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
        if(pCallback == NULL)
                return E_INVALIDARG;

        pCallback->AddRef();

        if(m_pCallback != NULL)
                m_pCallback->Release();

        m_pCallback = pCallback;

        return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
        if(out == NULL || *out == NULL || m_pCallback == NULL)
                return E_INVALIDARG;

        *out = m_pCallback;

        return S_OK;
}
```

Figure 69: A fix for the code shown in Figure 68

```
        axObject.MyProperty = new Object();

        var x = axObject.MyProperty();

        axObject.MyProperty = new Object();
```

Figure 70: JavaScript example with a release failure

trigger an integer overflow in the reference count variable. Let's try fixing it in the following example:

The above example adds a `Release()` call to correctly release any previously held objects, and so no memory leak occurs. Astute readers will notice that this code actually still has a stale pointer problem. The `get_MyProperty()` function doesn't add a reference to the interface being distributed back to the scripting engine. This can be problematic if the only reference to that interface is held by the plugin, and the plugin releases it. Consider the following JavaScript snippet:

This JavaScript code results in the following actions taking place:

1. `put_MyProperty()` retains the only reference to the object we created.

2. The `x` variable receives the `IDispatch` pointer, still there is only one copy of it

3. Setting `MyProperty` will cause the old object to be deleted, even though `x` still points to it.

6.2.1 Detecting COM Release Failures With Static Analysis

Detecting COM release failures is an opportunity for future research. There is a direct correlation to the Automatic Reference Counting technology employed by Apple's Xcode to prevent release failures of objective-C objects[16]. While this solution is not perfect, it has greatly reduced the number of memory leaks present in iPhone apps.

Chapter VII

CONCLUSION

This dissertation describes a number of vulnerabilities that arise due to the practice of building complex programming language constructs and frameworks on top of a primitive language that was never intended to support such complexity. Specifically, we focus on the adaptation of the C language by C++ and Microsoft's Component Object Model (COM) to support object orientation. The vulnerabilities that were identified are grouped into three categories: those that occur during object instantiation, those that occur during the lifetime of the object, and those that occur during the deletion of an object.

We demonstrated that when C++ objects are instantiated, security issues can occur when the object's constructor fails. It is difficult to check whether a C++ object was created properly since many of the elements of the object are unverifiable. As such, developers are forced to work with objects whose correctness cannot be guaranteed. We also demonstrated two issues that can occur during the instantiation of COM objects. The first is when a developer fails to explicitly increment the reference counter for the object. Because COM attempts to automatically reference count objects, it can find itself in a state where it improperly decrements the reference count. In this case, if the developer had not explicitly handled the reference counting, the object could be implicitly deleted without their knowledge. The second issue is similar to that which was show in C++. when a developer fails to call `VariantInit()` on a COM object, they later end up operating on an object that has not been properly initialized, resulting in serious security issues.

Throughout the lifetime of an object, there are security issues that can be encountered as well. Both C++ and COM suffer from design decisions that can easily lead to type confusion problems. In the case of C++, there is no runtime enforcement of type. With that, objects are treated as whatever the developer thought they were operating on at a given program point. Since there are many ways that a class pointer can be manipulated throughout the lifetime of the object, there is no way for the developer to guarantee the type of object they are using at any point. COM does attempt to implement type enforcement, but because this is built on a foundation that does not support it, what exists is highly imperfect. Developers (or attackers) can directly influence the type specifiers for a COM object, as well as other easily confused situations where developers can inadvertently treat one object as the wrong type. In both C++ and COM, type confusion can lead to exploitation as severe as arbitrary code execution.

COM suffers from additional security issues that can arise during the objects lifetime. First, copying objects at runtime is far more of a complex operation than it may seem on the surface. Failures to copy COM objects correctly can lead to developers operating on incomplete objects, or worse, vulnerabilities leading to arbitrary code execution. Second, the underlying security model for the loading of COM objects by applications is inherently flawed and easily bypassed.

As objects are deleted, yet further security issues may be encountered. Because C++ objects are build on C, the allocation and freeing of memory to contain objects is at the discretion of the developer. Developers will often aggressively try to free memory to avoid memory leaks. This, however, can lead to use-after-free conditions. In Section 6.1, we showed how this can be exploited. COM, on the other hand, attempts to automatically reference count and automatically release objects that are no longer in use. As described above, there are some fundamental flaws in this reference counting infrastructure since it is built on a platform that does not actually support it. With that, developers may aggressively increment the reference counter which can lead to integer wrapping, and thus unpredictable deletion of the object.

7.1 Summary of Results

In this section we summarize the results of the solutions that were developed to detect their corresponding security issues. In these cases, the solution was developed and tested for effectiveness. Each of the following subsections summarizes those results.

In Section 5.1.4, we describe a solution to detect C++ type confusion using static analysis. By employing the data flow analysis techniques documented in Section 5.1.4.2, we demonstrate that we can increase the effectiveness of existing static analysis techniques on compiled C++ code, as well as identify a class of vulnerability that is often overlooked by existing techniques. To test our decompilation framework, we created test programs each representing one of the four combinations of stack or heap object declaration and inline or explicit constructor. These programs were compiled without symbols and were provided to IDA Pro as input. In each case, the system was tested for its ability to resolve virtual function calls and to identify instances of type confusion. In all test cases, ORDA was able to resolve the virtual function calls correctly and identify cases where the indirect call referenced a function pointer outside the object's vtable.

In Section 5.4.5, we describe a solution for preventing trust transitivity problems in COM objects using runtime enforcement. We tested COMBlocker's effectiveness and measured the overhead it imposes on a standard desktop system. The first step in testing the effectiveness of the solution was to apply it to each of the applications that were found to be vulnerable in the section above. For each application that was shown to be vulnerable, COMBlocker presented the user with the appropriate dialog. This dialog box shows that the application attempted to load a control that is specifically denied by the defined policy. It also indicates that the instantiation of the object was prevented. COMBlocker was successfully able to prevent the attack described in Section 5.4.1 in Microsoft WordPad, Word, Excel, and PowerPoint as well as the ActiveX Control Test Container. Demonstrating the formal completeness of our solution is difficult. Our mechanism is helped by the fact that there are only a small number of publicly known means by which COM objects can be instantiated. Injecting COMBlocker at these points should logically prevent applications from circumventing policy enforcement. However, if applications can instantiate COM objects through other unknown means such as implementing their own APIs, these interfaces would also need to be modified and mediated.

A version of COMBlocker was created that logged the time required for each policy lookup encountered throughout the operation of an application. The test build was installed on a typical development workstation and gathered information for all of the COM object instantiations that occurred during a single day as part of a developer's normal work. The test workstation was a Windows XP SP3 machine with Office 2007, Internet Explorer 7, Firefox 3, Lotus Notes 8, Visual Studio 6, and several other commonly installed applications. During the course of the day, the behavior of the developer caused over 65,000 COM instantiations. Each of these recorded an average policy lookup time of 554µs to complete, with a 95% confidence interval of ±104µs. The variation in lookup time is largely due to the fact that consulting the killbit list in the registry is accomplished through a linear scan of the subkeys; it is not indexed. Testing shows that an average application incurs less than 10 policy lookups per user action. With that, each user action generates less than 5ms of delay due to COMBlocker. Another data point gathered in this test is that in general, Office applications and web browsers incurred a lower lookup time than core operating system components. When the data set is reduced to only Office applications and web browsers, the average lookup time drops to 104µs, with a 95% confidence interval of ±14.2µs. This indicates that if performance were an issue in implementing a system like COMBlocker, the scope of the protection could be reduced to only those applications that are more easily targeted in COM-based attacks.

In Section 6.1.3, we describe a solution for detecting C++ use-after-free conditions using static analysis. After implementing the Available Object Definition Analysis described in Section 6.1.3.1, we tested the framework in four ways. First, we tested it against a series of simple examples. These examples were used to first ensure the analysis was functioning correctly, and allowed for debugging in cases where something was missed. Details on the steps used can be found in Section 6.1.3.4. The second test was to inject vulnerabilities into into publicly available projects found on Github. By scanning these projects, we ensure that our analysis is able to function on real-world applications. The details of how vulnerabilities were injected into code and tested are covered in detail in Section 6.1.3.4. The results of these test are summarized in Table **??**. The third test was to determine whether the framework could identify a publicly disclosed vulnerability. We ran the analysis on tiptsf.dll

Table 7: Github projects with injected vulnerabilities

Project	Injected Vulns	Discovered Vulns
Leanify	1	1
libwebm	2	2
lifespan	1	1
MAPIEx	2	2
MonaServer	2	2
557-animator	2	2
BarsWF	1	1
easyrtc_ie_plugin	1	1
ElasticTabstopsForScintilla	1	1

and successfully identified the vulnerability disclosed as MS13-069 and CVE-2013-3205. The exact details of how this test was run and the results can be found in Section 6.1.3.4. The fourth and final test was to scan the System32 directory of an up-to-date installation of Windows 7 to see how many use-after-free conditions could be identified. In scanning 652 libraries, we were able to identify 127 use-after-free conditions. Section 6.1.3.4 provides details on how this test was run. Table **??** summarizes the results.

7.2 Future Work

In this section we summarize those portions of the dissertation which propose new solutions, but have not been thoroughly developed or tested. Sections 4.1.1 and 4.3.1 discuss future research that is focused on identifying failures in the instantiation of C++ and COM objects. Sections 4.2.2 and 6.2.1 discuss potential solutions for detecting failure to retain and release COM objects. Here we propose researching opportunities similar to Automatic Reference Counting used in Apple's Xcode. Sections 5.2.4 and 5.3.2 discuss areas of future research for detecting security issues that arise during the lifetime of COM objects.

7.3 Final Thoughts

In this dissertation, we have shown how building high-level programming constructs on a primitive language has led to security issues. Previous attempts to analyze binary code that was developed at these higher-levels have proven to struggle. This is due to the fact that they still operate at the lowest levels of context. By elevating the context of the analyses

Table 8: DLLs containing use-after-free vulnerabilities

Library	Potential Use-after-free Conditions
ActionCenterCPL.dll	5
AdmTmpl.dll	16
bidispl.dll	3
cabview.dll	12
certcli.dll	1
cewmdm.dll	8
cnvfat.dll	1
comdlg32.dll	1
comsnap.dll	9
credui.dll	6
cscapi.dll	2
dataclen.dll	2
devenum.dll	3
devmgr.dll	2
eapp3hst.dll	2
eappcfg.dll	4
Faultrep.dll	2
FirewallAPI.dll	1
fontext.dll	5
fundisc.dll	4
gpprnext.dll	1
gpedit.dll	3
iasrad.dll	1
icsigd.dll	1
imagehlp.dll	1
ipsmsnap.dll	23
itss.dll	8

closer to that of the original language, we have been able to address the higher-level security issues. We have shown that these analyses can be highly effective in detecting and remedying security issues that would not have been addressable at a lower level of context. With this work, we expect that security analysts will be able to identify security issues in existing applications that were previously undetectable.

Appendix A

The following code is the LLVM intermediate representation generated for the binary code shown in Figure 39b.

```
; ModuleID = 'classbug2.bc'

%0 = type { i32 (...)**, i32 }

@dummyClass = weak_odr constant [100 x i32 (...)*] [i32 (...)*
@dummyFunc, i32 (...)* @dummyFunc, i32 (...)* @dummyFunc, i32 (...)*
@dummyFunc, i32 (...)* @dummyFunc, i32 (...)* @dummyFunc, i32 (...)*
@dummyFunc, i32 (...)* @dummyFunc] ; <[100 x i32 (...)*]*> [#uses=0]

@off_402138 = weak_odr constant [4 x i32 (...)*] [i32 (...)* bitcast
(i32 (i32)* @sub_4010B0 to i32 (...)*), i32 (...)* bitcast (i32 ()*
@sub_401080 to i32 (...)*), i32 (...)* bitcast (i32 ()* @nullsub_1 to
i32 (...)*), i32 (...)* bitcast (i32 ()* @sub_401090 to i32 (...)*)] ;
<[4 x i32 (...)*]*> [#uses=1]

@off_40214C = weak_odr constant [2 x i32 (...)*] [i32 (...)* bitcast (i32
(i32)* @sub_4010B0 to i32 (...)*), i32 (...)* bitcast (i32 ()*
@sub_4010C0 to i32 (...)*)] ; <[2 x i32 (...)*]*> [#uses=1]

declare i32 @dummyFunc(...)

define i32 @_wmain(i32 %Arg_esi) {
"401000":
  %0 = call i8* @"??2@YAPAXI@Z"() ; <i8*> [#uses=2]
  %Class1 = alloca %0* ; <%0**> [#uses=2]
  %Class0 = alloca %0* ; <%0**> [#uses=2]
  %1 = ptrtoint i8* %0 to i32 ; <i32> [#uses=1]
  %2 = icmp eq i32 %1, 0 ; <i1> [#uses=1]
  br i1 %2, label %"401019", label %"401010"
```

```
"401010": ; preds = %"401000"
  %3 = call i32 @sub_401070() ; <i32> [#uses=1]
  %4 = inttoptr i32 %3 to %0** ; <%0**> [#uses=2]
  %5 = bitcast %0** %4 to i8* ; <i8*> [#uses=1]
  %6 = load %0** %4 ; <%0*> [#uses=1]
  store %0* %6, %0** %Class0
  br label %"40101b"

"401019": ; preds = %"401000"
  %7 = alloca i32 ; <i32*> [#uses=2]
  store i32 0, i32* %7
  %8 = load i32* %7, align 4 ; <i32> [#uses=1]
  %9 = inttoptr i32 %8 to %0** ; <%0**> [#uses=1]
  br label %"40101b"

"40101b": ; preds = %"401019", %"401010"
  %10 = phi i8* [ %0, %"401019" ], [ %5, %"401010" ] ; <i8*> [#uses=0]
  %11 = phi %0** [ %9, %"401019" ], [ %Class0, %"401010" ] ;
        <%0**> [#uses=2]
  %12 = call i8* @"??2@YAPAXI@Z"() ; <i8*> [#uses=2]
  %13 = ptrtoint i8* %12 to i32 ; <i32> [#uses=1]
  %14 = icmp eq i32 %13, 0 ; <i1> [#uses=1]
  br i1 %14, label %"401032", label %"401029"

"401029": ; preds = %"40101b"
  %15 = call i32 @sub_4010A0() ; <i32> [#uses=1]
  %16 = inttoptr i32 %15 to %0** ; <%0**> [#uses=2]
  %17 = bitcast %0** %16 to i8* ; <i8*> [#uses=1]
  %18 = load %0** %16 ; <%0*> [#uses=1]
  store %0* %18, %0** %Class1
  br label %"401034"

"401032": ; preds = %"40101b"
  %19 = alloca i32 ; <i32*> [#uses=2]
  store i32 0, i32* %19, align 4
  %20 = load i32* %19, align 4 ; <i32> [#uses=1]
  %21 = inttoptr i32 %20 to %0** ; <%0**> [#uses=1]
  br label %"401034"

"401034": ; preds = %"401032", %"401029"
  %22 = phi i8* [ %12, %"401032" ], [ %17, %"401029" ] ;
        <i8*> [#uses=0]
  %23 = phi %0** [ %21, %"401032" ], [ %Class1, %"401029" ] ;
        <%0**> [#uses=3]
  %24 = load %0** %11, align 4 ; <%0*> [#uses=1]
  %25 = getelementptr %0* %24, i32 0, i32 0 ; <i32 (...)***> [#uses=1]
  %26 = load i32 (...)*** %25, align 4 ; <i32 (...)**> [#uses=1]
  %27 = getelementptr i32 (...)** %26, i32 0 ; <i32 (...)**> [#uses=1]
  %28 = load i32 (...)** %27 ; <i32 (...)*> [#uses=1]
  %29 = bitcast i32 (...)* %28 to void ()* ; <void ()*> [#uses=1]
  call void %29()
  %30 = load %0** %11, align 4 ; <%0*> [#uses=1]
  %31 = getelementptr %0* %30, i32 0, i32 0 ; <i32 (...)***> [#uses=1]
  %32 = load i32 (...)*** %31, align 4 ; <i32 (...)**> [#uses=1]
  %33 = getelementptr i32 (...)** %32, i32 1 ; <i32 (...)**> [#uses=1]
  %34 = load i32 (...)** %33 ; <i32 (...)*> [#uses=1]
  %35 = bitcast i32 (...)* %34 to void ()* ; <void ()*> [#uses=1]
  call void %35()
  %36 = load %0** %23, align 4 ; <%0*> [#uses=1]
  %37 = getelementptr inbounds %0* %36, i32 0, i32 0 ;
        <i32 (...)***> [#uses=1]
  %38 = load i32 (...)*** %37, align 4 ; <i32 (...)**> [#uses=1]
  %39 = getelementptr i32 (...)** %38, i32 0 ; <i32 (...)**> [#uses=1]
  %40 = load i32 (...)** %39 ; <i32 (...)*> [#uses=1]
  %41 = bitcast i32 (...)* %40 to void ()* ; <void ()*> [#uses=1]
  call void %41()
  %42 = load %0** %23, align 4 ; <%0*> [#uses=1]
  %43 = getelementptr inbounds %0* %42, i32 0, i32 0 ;
        <i32 (...)***> [#uses=1]
  %44 = load i32 (...)*** %43, align 4 ; <i32 (...)**> [#uses=1]
  %45 = getelementptr i32 (...)** %44, i32 1 ; <i32 (...)**> [#uses=1]
  %46 = load i32 (...)** %45 ; <i32 (...)*> [#uses=1]
  %47 = bitcast i32 (...)* %46 to void ()* ; <void ()*> [#uses=1]
  call void %47()
  %48 = load %0** %23, align 4 ; <%0*> [#uses=1]
  %49 = getelementptr inbounds %0* %48, i32 0, i32 0 ;
        <i32 (...)***> [#uses=1]
  %50 = load i32 (...)*** %49, align 4 ; <i32 (...)**> [#uses=1]
  %51 = getelementptr i32 (...)** %50, i32 3 ; <i32 (...)**> [#uses=1]
  %52 = load i32 (...)** %51 ; <i32 (...)*> [#uses=1]
  %53 = bitcast i32 (...)* %52 to void ()* ; <void ()*> [#uses=1]
  call void %53()
  %54 = alloca i32 ; <i32*> [#uses=2]
  store i32 0, i32* %54, align 4
  %55 = load i32* %54, align 4 ; <i32> [#uses=0]
  br label %return

return: ; preds = %"401034"
  ret i32 0
}

define i32 @sub_401070() {
"401070":
  %0 = alloca %0* ; <%0**> [#uses=2]
  %1 = load %0** %0, align 4 ; <%0*> [#uses=1]
  %2 = getelementptr inbounds %0* %1, i32 0, i32 0 ;
        <i32 (...)***> [#uses=1]
  %3 = getelementptr inbounds [4 x i32 (...)*]*
                @off_402138, i32 0, i32 0 ; <i32 (...)**> [#uses=1]
  store i32 (...)** %3, i32 (...)*** %2
  %4 = load %0** %0, align 4 ; <%0*> [#uses=1]
  %5 = getelementptr inbounds %0* %4, i32 0, i32 1 ;
        <i32*> [#uses=1]
  store i32 0, i32* %5, align 4
  br label %return
```

```
return: ; preds = %"401070"
  ret i32 0
}

define i32 @sub_4010A0() {
"4010a0":
  %0 = alloca %0* ; <%0**> [#uses=2]
  %1 = load %0** %0, align 4 ; <%0*> [#uses=1]
  %2 = getelementptr inbounds %0* %1, i32 0, i32 0 ;
        <i32 (...)***> [#uses=1]
  %3 = getelementptr inbounds [2 x i32 (...)*]*
                @off_40214C, i32 0, i32 0 ; <i32 (...)**> [#uses=1]
  store i32 (...)** %3, i32 (...)*** %2
  %4 = load %0** %0, align 4 ; <%0*> [#uses=1]
  %5 = getelementptr inbounds %0* %4, i32 0, i32 1 ; <i32*> [#uses=1]
  store i32 0, i32* %5, align 4
  br label %return

return: ; preds = %"4010a0"
  ret i32 0
}
```

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Adobe. Security update available for Adobe Flash Player. `http://www.adobe.com/support/security/bulletins/apsb11-07.html`, April 2011.

[3] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[4] Vinod Anupam and Alain Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *Proceedings of the USENIX Security Symposium*, 1998.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29:290–301, June 1994.

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1996.

[7] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *ACM Conference on Programming Language Design and Implementation*, June 2005.

[8] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *in PLDI 06*, pages 158–168. ACM Press, 2006.

[9] Bugscam. Bugscam IDC Package. `http://bugscam.sourceforge.net/`.

[10] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.

[11] Juan Caballero, Gustavo Greico, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

[12] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Portland, Oregon, 1994.

[13] Hao Chen and David Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2002.

[14] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

[15] Clang. `http://clang.llvm.org/`.

[16] Clang. "automatic reference counting". Clang 3.8 Documentation. `http://clang.llvm.org/docs/AutomaticReferenceCounting.html`.

[17] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[18] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *In European Symposium on Programming*, 2007.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Los Angeles, California, 1977.

[20] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[21] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[22] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *In ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2003*, pages 69–80. ACM Press, 2003.

[23] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, 2009.

[24] Mark Dowd, Ryan Smith, and David Dewey. Attacking Interoperability. Proceedings of Black Hat 2009, July 2009. `https://media.blackhat.com/bh-usa-09/video/DOWD/BHUSA09-Dowd-AtkInterop-VIDEO.mov`.

[25] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*, 2009.

[26] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, January 2004.

[27] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.

[28] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*, May 1999.

[29] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[30] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[31] Timothy Fraser, Nick L Petroni, and William A Arbaugh. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2006.

[32] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[33] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*, 2006.

[34] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the Internation Conference on Software Engineering (ICSE)*, 2007.

[35] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE, 2014.

[36] Google. Gcc vtable security hardening proposal. `http://gcc.gnu.org/ml/gcc-patches/2012-11/txt00001.txt`.

[37] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[38] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, May-June 2011.

[39] Hey-Rays. `http://www.hex-rays.com/`.

[40] Galen Hunt and Doug Brubacher. Detours: binary interception of win32 functions. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[41] Nayeem Islam, Rangachari Anand, Trent Jaeger, and Josyula R. Rao. A flexible security system for using internet content. *IEEE Software*, 14:52–59, 1997.

[42] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *ACM Transactions on Information and System Security (TISSEC)*, 7(2), 2004.

[43] Trent Jaeger, Aviel D. Rubin, and Atul Prakash. Building systems that flexibly control downloaded executable context. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 1996.

[44] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks.

[45] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c.

[46] Stephen Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1977.

[47] Richard W M Jones, Paul H J Kelly, Most C, and Uncaught Errors. Backwards-compatible bounds checking for arrays and pointers in c programs. In *in Distributed Enterprise Applications. HP Labs Tech Report*, pages 255–283, 1997.

[48] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[49] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, 2015.

[50] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for java. *ACM SIGPLAN Notices*, 36(11):367–380, 2001.

[51] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, volume 49, pages 227–238. ACM, 2014.

[52] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, and Ruth C. Taylor. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the National Information Systems Security Conference*, 1998.

[53] Dahlia Malkhi and Michael Reiter. Secure execution of java applets using a remote playground. *IEEE Transactions on Software Engineering*, 27(12):1197–1209, 2000.

[54] Microsoft. Safe Initialization and Scripting for ActiveX Controls. Microsoft Developer Network. `http://msdn.microsoft.com/en-us/library/aa751977(VS.85).aspx`.

[55] Microsoft. static_cast Operator. Microsoft Developer Network. `http://msdn.microsoft.com/en-us/library/c36yw7x9%28v=vs.80%29.aspx`.

[56] Microsoft. Working with the AppInitDLLs registry value. Microsoft Support, November 2006. `http://support.microsoft.com/kb/197571`.

[57] Microsoft. Microsoft Security Advisory (953839). Microsoft TechNet, August 2008. `http://www.microsoft.com/technet/security/advisory/953839.mspx`.

[58] Microsoft. Microsoft Security Advisory (956391). Microsoft TechNet, October 2008. `http://www.microsoft.com/technet/security/advisory/956391.mspx`.

[59] Microsoft. How to stop an ActiveX control from running in Internet Explorer. Microsoft Support Center, August 2009. `http://support.microsoft.com/kb/240797`.

[60] Microsoft. Microsoft Security Advisory (960715). Microsoft TechNet, February 2009. `http://www.microsoft.com/technet/security/advisory/960715.mspx`.

[61] Microsoft. Microsoft Security Advisory (969898). Microsoft TechNet, June 2009. `http://www.microsoft.com/technet/security/advisory/969898.mspx`.

[62] Microsoft. Microsoft Security Bulletin MS09-032. Microsoft TechNet, July 2009. `http://www.microsoft.com/technet/security/bulletin/ms09-032.mspx`.

[63] Microsoft. Microsoft Security Bulletin MS09-034. `http://www.microsoft.com/technet/security/bulletin/ms09-034.mspx`, July 2009.

[64] Microsoft. Microsoft Security Bulletin MS09-035. Microsoft TechNet, July 2009. `http://www.microsoft.com/technet/security/bulletin/Ms09-035.mspx`.

[65] Microsoft. Microsoft Security Bulletin MS10-017. `http://www.microsoft.com/technet/security/bulletin/MS10-017.mspx`, March 2010.

[66] Microsoft Support. You are prompted to grant permission for ActiveX Controls when you open an Office XP or Office 2003 document. Microsoft Support Center, October 2007. `http://support.microsoft.com/default.aspx?scid=kb;en-us;827742`.

[67] Matthew R Miller, Kenneth D Johnson, and Timothy William Burrell. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities, March 25 2014. US Patent 8,683,583.

[68] Eric Moretti, Gilles Chanteperdrix, and Angel Osorio. New algorithms for control-flow graph structuring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 184–, Washington, DC, USA, 2001. IEEE Computer Society.

[69] George C. Necula, Jeremy Condit, Matthew Harren, Scott Mcpeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27:2005, 2005.

[70] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code, 2002.

[71] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[72] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[73] Hemant Pande and Barbara Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Symposium on Static Analysis (SAS)*, 1996.

[74] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Proceedings of the 6th USENIX C++ Technical Conference*, 1994.

[75] Pete Becker. Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf`.

[76] David J. Roth and David S. Wise. One-bit counts between unique and sticky. In *ACM SIGPLAN Notices*, pages 49–56. ACM Press, 1998.

[77] Paul Sabanal and Mark Yason. Reversing C++. In *Proceedings of BlackHat DC*, 2007.

[78] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[79] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[80] Asia Slowinska, Traiain Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[81] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, 2008.

[82] Sparse. `https://sparse.wiki.kernel.org/index.php/Main_Page`.

[83] Splint. `http://splint.org/`.

[84] Bjarne Stroustrup. *The Design and Evolution of C++*. Pearson Education, 1994.

[85] Caroline Tice. Extended Abstract: Improving Function Pointer Security for Virtual Method Dispatches. GNU Tools Cauldron Workshop, 2012.

[86] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC)*, 2000.

[87] Wikipedia. static_cast. `http://en.wikipedia.org/wiki/Static_cast`.

[88] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the European Software Engineering Conference (ESEC)*, 2003.

[89] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2003.

[90] Bin Zeng, Gang Tan, and Greg Morrisett. Combining controlflow integrity and static analysis for efficient and validated data sandboxing. In *In Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS) (2011*, page 29.

[91] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen Mccamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *in Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013*, pages 559–573.

[92] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries.

[93] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 2002.