Syracuse University
SURFACE

Electrical Engineering and Computer Science Technical Reports

L.C. Smith College of Engineering and Computer Science

2-3-2012

Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis

Mingwei Zhang
University of Singapore

Aravind Prakash
Syracuse University

Xiaolei Li
National University of Singapore

Zhenkai Liang
National University of Singapore

Heng Yin
Syracuse University, heyin@syr.edu

Follow this and additional works at: http://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation
Zhang, Mingwei; Prakash, Aravind; Li, Xiaolei; Liang, Zhenkai; and Yin, Heng, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis" (2012). *Electrical Engineering and Computer Science Technical Reports.* Paper 26.
http://surface.syr.edu/eecs_techreports/26

This Report is brought to you for free and open access by the L.C. Smith College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Department of Electrical Engineering and Computer Science

Technical Report

SYR-EECS-2012-01 February 3, 2012

Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis

Mingwei Zhang School of Computing, National University of Singapore

Aravind Prakash arprakas@syr.edu, Syracuse University

Xiaolei Li School of Computing, National University of Singapore

Zhenkai Liang School of Computing, National University of Singapore

Heng Yin heyin@syr.edu, Syracuse University

ABSTRACT: Software exploits are one of the major threats to the Internet security. A large family of exploits works by corrupting memory of the victim process to execute malicious code. To quickly respond to these attacks, it is critical to automatically diagnose such exploits to find out how they circumvent existing defense mechanisms. Because of the complexity of the victim programs and sophistication of recent exploits, existing analysis techniques fall short: they either miss important attack steps or report too much irrelevant information. In this paper, based on the observation that the key steps in memory corruption exploits often involve pointer misuses, we propose a novel solution, PointerScope, to use type inference on binary execution to detect the pointer misuses induced by an exploit. These pointer misuses highlight the important attack steps of the exploit, and therefore convey valuable information about the exploit mechanisms. Our approach complements dependency-based solutions to perform more comprehensive diagnosis of sophisticated memory exploits. We prototyped PointerScope and evaluated it using real-world exploit samples and demonstrated that PointerScope can successfully capture the key attack steps, which significantly facilitates attack response.

KEYWORDS: Memory Forensics, Type Propagation, Pointer violation, Pointerscope, Memory exploit

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis

Mingwei Zhang[1], Aravind Prakash[2], Xiaolei Li[1], Zhenkai Liang[1], and Heng Yin[2]
[1]*School of Computing, National University of Singapore*
[2]*Department of Computer Science, Syracuse University*

Abstract

Software exploits are one of the major threats to the Internet security. A large family of exploits works by corrupting memory of the victim process to execute malicious code. To quickly respond to these attacks, it is critical to automatically diagnose such exploits to find out how they circumvent existing defense mechanisms. Because of the complexity of the victim programs and sophistication of recent exploits, existing analysis techniques fall short: they either miss important attack steps or report too much irrelevant information. In this paper, based on the observation that the key steps in memory corruption exploits often involve pointer misuses, we propose a novel solution, PointerScope*, to use type inference on binary execution to detect the pointer misuses induced by an exploit. These pointer misuses highlight the important attack steps of the exploit, and therefore convey valuable information about the exploit mechanisms. Our approach complements dependency-based solutions to perform more comprehensive diagnosis of sophisticated memory exploits. We prototyped PointerScope and evaluated it using real-world exploit samples and demonstrated that PointerScope can successfully capture the key attack steps, which significantly facilitates attack response.*

1 Introduction

Software exploits are one of the major threats to the Internet security. A large family of exploits works by corrupting memory of the victim process to execute malicious code. To quickly respond to such attacks, defenders critically need techniques that can automatically diagnose an exploit and understand the inner-working of an exploit.

The main difficulty in diagnosing memory-corruption exploits is caused by the constantly-evolving attack techniques. Memory-corruption exploits have existed for decades, but recent attack techniques are getting more and more sophisticated. For example, due to the wide adoption of memory-corruption prevention mechanisms, such as address-space-layout randomization (ASLR) [3, 9], attackers are no longer able to *directly* execute malicious code they have injected. They need to resort to additional techniques to bypass the defense mechanisms and locate their malicious code. As a result, recent memory-corruption attacks often consist of a sequence of *key steps*. A major challenge to attack diagnosis is to get the complete picture of the attack, including all steps from the initial vulnerability exploit to the malicious code execution.

Existing techniques of attack diagnosis mainly focus on a single step of the attacks. For example, some techniques [22, 37] perform analysis on a memory snapshot at the moment when the attack is detected. The effectiveness of such memory analysis is limited by available information in the snapshot, because the evidence of previous attack steps may have already been destroyed before the attack is detected.

It has been observed that traditional memory exploits often involve misuse of pointers [13]. By analyzing several recent memory-corruption exploits, we found that this observation can be extended to most of the key steps in a sophisticated attack, which misuses other types of data as pointers, especially pointers that control program execution, such as return addresses and virtual function pointers. Based on this observation, we propose a novel technique, called PointerScope, for automated exploit diagnosis by detecting and analyzing pointer misuses. That is, from the execution of a binary executable, we automatically infer the type of each of registers and memory locations. As pointer misuse is one of the most important characteristics of key attack steps, our approach detects misuses as type conflicts when other types of data are used as pointers, especially control pointers. Then we identify the instructions and operands involved in the type conflict, and understand how an exploit happens and how it circumvents existing defense mechanisms.

To evaluate the efficacy of our approach, we have implemented a prototype system of PointerScope and evaluated it using real-world memory-corruption exploits. The experiments demonstrate that PointerScope detected pointer mis-

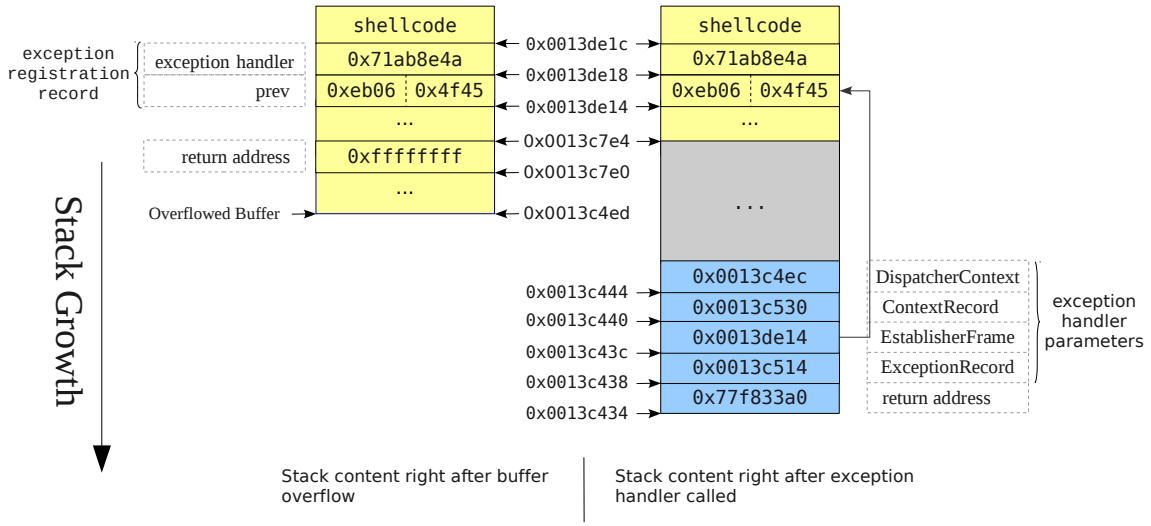

Figure 1. Stack layout in an SEH exploit.

uses as the key attack steps and highlighted a small number of instructions from a trace of multi-million instructions.

In summary, we have made the following main contributions:

- We propose to identify and analyze pointer misuses as a generic approach for diagnosing sophisticated memory-corruption exploits. Our approach complements existing dependency-based solutions, and diagnoses the exploits to reveal details of the attacks' key steps.

- We define a small type system on x86 architecture and devise a type inference algorithm to resolve type information during the binary execution. Pointer misuses can then be detected as type conflicts.

- We implement a prototype system of PointerScope and evaluate it using seven real-world exploit samples, demonstrating the capability and effectiveness of this tool.

Paper Organization The remainder of this paper is organized as follows. Section 2 uses an example of sophisticated memory-error exploits to motivate our solution. Section 3 details the design of PointerScope. Section 4 describes our prototype implementation challenges and Section 5 discusses the experimental results. Section 6 presents the limitations of the current system and future work. We discuss related work in Section 7. Section 8 concludes the paper.

2 Background

In this section, we describe a motivating example of sophisticated memory-error exploits and discuss the need for new solutions for diagnosing the key steps of such exploits.

A Motivating example. Structured Exception Handling (SEH) [27] is a mechanism for handling program exceptions on Windows. Under this mechanism, a thread registers the handlers for its exceptions, and Windows maintains the registered exception handlers of a thread in a list of *exception registration records* in the thread's Thread Information Block, which is located at the bottom of the stack. Each record has a function pointer field, pointing to the exception handler function. Windows requires that exception handlers have the following function signature:

```
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

When an exception occurs, Windows locates the exception registration record, prepares all the parameters on top of the stack, and invokes the exception handler. Among the parameters to the exception handler, the second parameter, `EstablisherFrame`, is the address of the current exception registration record.

The SEH mechanism has been abused by attackers as a key step to circumvent stack protections [26]. Because of the deployment of defense mechanisms against memory-error exploits, such as address-space layout randomization,

attackers cannot easily locate the malicious code they inject into the victim process. In this attack, the SEH mechanism is leveraged to allow attackers to reliably locate the injected code. We illustrate the attack in Figure 1. First, the attack overflows an unbounded buffer to corrupt the stack all the way to the bottom of the stack, overwriting both the return address and the exception registration records. The overwritten return address is set to an invalid address (e.g., *0xffffffff*), which triggers an exception when the corresponding function returns, illustrated on the left-hand part of Figure 1. In response to this exception, the Windows SEH mechanism prepares the parameters to the exception handler and stores them on top of the stack, illustrated on the right-hand part of Figure 1. Note that the second parameter (`EstablisherFrame` at the address *0x0013c43c*) points to the current exception registration record, which is at the address *0x0013de14*. To take advantage of this pointer, the attacker overwrites the function pointer (at the address *0x0013de18*) to the exception handler with an address pointing to an instruction sequence in memory: `pop %esi; pop %ecx; ret` (we call this sequence as a ***PPR slice*** in short). In this example, this variable value is *0x71ab8e4a*, which points to a PPR slice in the library `ws2_32.dll`. When the call to the exception handler executes this PPR slice, the two items on the stack top, the return address and `ExceptionRecord`, are popped out, and the `EstablisherFrame` parameter is used as the return address, leading the control flow to the address *0x0013de14*, which is the start of the exception registration record. The attacker has prepared a short jump instruction (`jmp $8`, represented as binary *0xeb06*), which redirects the control follow to the shellcode placed at the address *0x0013de1c*. Note that in this attack, the attacker does not need to locate the injected shellcode. It is done by exploiting the SEH mechanism.

This attack example has the following key steps: 1) an exception is triggered by returning to an invalid memory address; 2) a corrupted exception routine is invoked in response; 3) a PPR slice is executed, which transfers control to injected code on the stack; and 4) the injected code is executed.

Challenges to taint-based analysis. This attacks illustrated some of the difficulties faced by taint-based approaches in attack diagnosis. Using taint analysis, we can observe that the overwritten return address at address *0x0013c7e0* is tainted by external inputs. However, the control transfer in the third step does not get its target address from attackers. Instead, it is the SEH handling mechanism that pushes the target address onto the stack, so the address is not affected by external inputs from attackers. Therefore, taint analysis techniques cannot effectively identify the third key step in the above example, although they have no problem in identifying the initial exploit (the first step), as well as the execution of injected code (the fourth step). Understanding the complete picture of this attack is important in identifying the weakness of the exception handling mechanism, leading to solutions that can prevent the class of attacks. For example, Windows updates its exception handling mechanism to SafeSEH to prevent this type of attacks.

In general, dependency analysis, such as taint analysis [25] and backward slicing [5], has demonstrated its power and effectiveness in memory exploit detection and diagnosis. However, to identify the key steps of an exploit and reason about the causal relationships among them, we need to take into account all kinds of dependency relationships, including direct data dependency, indirect data dependency, and control dependency: Direct data dependency considers data movement from a location to another; Indirect data dependency incorporates the dependency between a memory address and the memory content pointed to by this index; Control dependency, on the other hand, models the dependency between a control decision and a data variable that is set based on this control decision. However, conservatively considering all these dependencies would extract enormous amount of dependencies, most of which are actually irrelevant to the exploit. Therefore, in reality, all the existing binary analysis techniques only consider a subset of these dependencies. For example, taint analysis mainly considers direct data dependency, and may selectively enable indirect data dependency using heuristic policies. A recent technique [17] expands this type of approaches by tracking a subset of control dependency. In consequence, important key steps and causal relationships may be unfortunately excluded from the diagnosis report.

Our observation. Exploits to computer systems involve using program features in an unintended way. For example, in memory-error exploits, attacks often rely on pointer misuses. Considering the above example, in the first key step, an exception is triggered by using an invalid address as return address (control pointer). In the second key step, the process uses an overflowed function pointer to the exception handler. In the third step, the last instruction of the PPR slice, `ret`, takes the value at the stack as the return address, which is the second parameter to the exception handler. This argument has been pushed onto stack as an `EstablisherFrame` pointer, pointing to the current exception registration record.

Pointer misuse, a crucial characteristic in these key steps for a successful attack, is the result of the flexibility of the x86 instruction set. However, the programs produced by compilers use the instructions in a rather consistent way. In particular, the pointer values, especially control pointers, are rarely derived from data of other types. Following this observation, we aim to detect and analyze pointer misuses by summarizing the common usage of instructions and detect misuses of pointers.

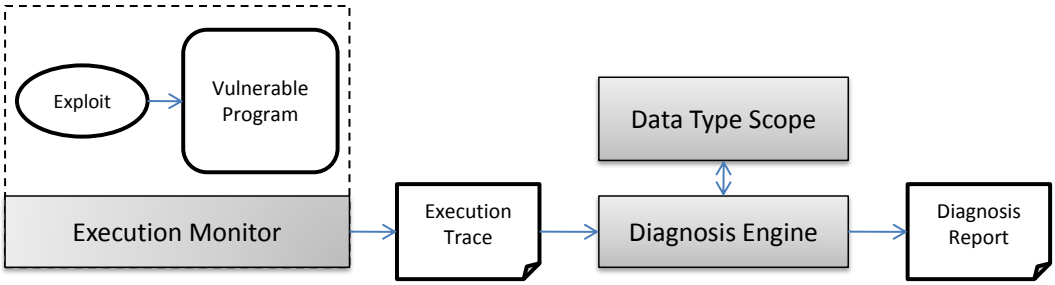

Figure 2. Overview of PointerScope. Core components of PointerScope are illustrated in dark nodes.

3 Design of PointerScope

Given an exploit and its corresponding vulnerable program, the primary goal of PointerScope is to highlight the exploit's *key steps* in the huge amount of instructions and help security analysts to understand the overall attack mechanisms.

3.1 Approach overview

Figure 2 depicts the overview of our approach, where the core components of PointerScope are shown in dark color. The *execution monitor* dynamically runs the vulnerable program with the given exploit, and collects a detailed execution trace. It also collects available auxiliary information, such as the memory locations of loaded modules and symbols file (e.g., pdb files for Windows DLLs), to facilitate further analysis. The *diagnosis engine* processes the execution trace and auxiliary information, and uses the *data type scope* to infer data types of registers and memory locations during the execution of the vulnerable program. The data type scope detects pointer misuses as type conflicts. Based on the result of the data type scope, the diagnosis engine generates the diagnosis report by extracting the causal dependency relationships between the conflicts.

The main challenge faced by PointerScope is to infer data types from instructions, and to define the consistency rules among such types. Note that unlike recent solutions in type inference on binaries [21, 23], the purpose of PointerScope is not to reveal the rich program information available in high-level languages. Instead, it aims to facilitate exploit diagnosis by detecting type conflicts on instructions.

3.2 Detecting pointer misuse

In order to detect pointer misuses, we need to get the type information of instruction operands to distinguish pointers and other data. For this purpose, we define the data types and the set of constraint rules for x86 instructions. We focus on the 32-bit x86 instruction set in this paper.

Type	Description
INT	Integer
CTR	Control pointer
DTR	Data pointer
OTR	Types other than integer and pointer
PTR	pointer, supertype of CTR and DTR
ANY	can be any type
NTR	Non-pointer
CFL	conflict

Table 1. Types defined in PointerScope.

Type definitions Table 1 lists the types defined in PointerScope. The top portion of Table 1 lists the primitive types: *INT* for integers, *CTR* for control pointers, *DTR* for data pointers, and *OTR* for other types. *Integers* include index, offset, counter and so on; *Control pointers* are memory addresses for code; *Data pointers* are memory addresses for data; *Others* are for all other types of variable, which we do not need to distinguish for the application of pointer misuse detection. The bottom portion of Table 1 lists aggregate types, which are combined from primitive types. Aggregate types are used to group types that can be treated similarly into a super-type, so that the type system in PointerScope will not be unnecessarily complex.

Because the objective of PointerScope is to help exploit diagnosis, we keep the type system as simple as possible to avoid overwhelming conflicts that may confuse security analysts.

Instruction type constraints The type constraints posed by instructions on their operands can be classified into two categories. First, some instructions *generate* constraints to the types of its operands. For example, multiplications (`mul`/`imul`) and divisions (`div`/`idiv`) operate on integers. Therefore, if any of these instructions is executed, we can infer that the operands must be *INT*. Second, certain instructions (e.g., data transfer instructions `mov`, `push`, and `pop`) *propagate* type from the source operand to the destination operand.

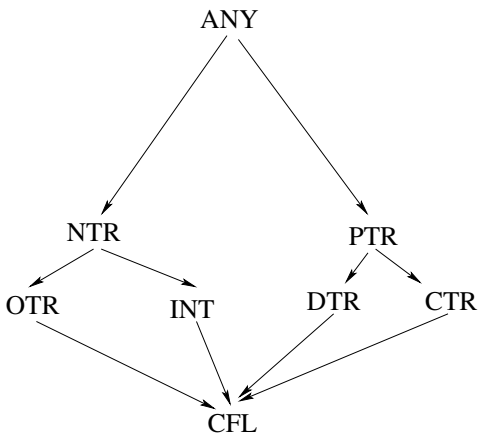

Figure 3. Relationship among types.

We studied the x86 instruction set and defined the type constraints of common instructions. We summarize the constraints that common general purpose x86 instructions put on their operands in Table 4 of the Appendix.

Note that the task of deciding an instruction's type constraint is complicated by a few factors. First, an instruction's behavior may depend on the value of its operands. For example, the `cmove` instruction conditionally moves data from its source operand to its destination operand. Second, some instructions have implicit operands. For example, the `call` instruction creates a return address on the stack, which is also a control pointer.

When an instruction takes two typed operands as input, we need to decide the type of the output operand. For typical scenarios, the result is straightforward, such as adding two integers and adding a pointer and an integer. The main challenge is caused by the special usage of instructions for purposes such as performance optimization. We summarize the common cases by analyzing common instruction patterns, and define the output type of an instruction for such special cases. For example, bit operations `and`, `xor`, `or` are commonly used in special ways: `and %eax, %eax` and `or %eax, 0` are used for zero test, while `xor %eax, %eax` is used to set `%eax` to zero. We summarize them in Appendix. As another example, `lea $0x8(%eax, %ebx), %ecx` is not always for memory access. Instead, it is often used as an efficient way to compute `%eax+%ebx+$0x8` and store the result in the register `ecx`. We discuss more about such cases in Section 4.

External type constraints Type constraints from external sources are also important. Many APIs and system calls are documented, so type definitions in their argument lists can be used to create type constraints.

```
char *strncpy(char *dst, const char *src,
              size_t num);
```

For example, the above function prototype indicates that the first two arguments are data pointers and the third argument is a 32-bit integer. If available, we may also use type information in debugging symbols to create type constraints.

Another source of type information is the binary executable itself. Meta-data in the binary reveal operand type information. Particularly, the relocation table lists the addresses of constant numbers that need to be updated during relocation. These constant numbers are address references either to a data variable or a function entry. So we resolve them as *PTR*. On the contrary, the constant numbers that do not appear in the relocation table are resolved as *NTR*. The only exception is 0, which can be either a NULL pointer or a zero value, and its address does not appear in the relocation table. So it has to be *ANY* for its initial type.

Type inference and conflict detection The algorithm we use for type inference is based on the classic type inference algorithm for a high-level programming language, *Algorithm W* [15]. We adapt this algorithm to suit our purpose of exploit diagnosis.

First, in binary programs, we treat memory locations and registers as variables. Unlike variables in a high-level programming language, variables in binary programs are frequently reused. Registers and local stack locations are often used to store different types of data at different execution stages. Therefore, for a destination operand, we cannot accumulate its constraints and detect type conflicts during the binary execution. Instead, we always perform a *strong* update on the destination operand with the most precise type at each execution moment. Type refinement and conflict detection are only performed on source operands. In other words, type refinement and conflict detection are performed on variable reads, not on variable writes. This strategy is also used to deal with `union`, where a field may have multiple types under different execution contexts.

Second, in the Algorithm W, all the type constraints of variables are accumulated throughout the entire program, and then the most precise type is resolved for each variable. In our case, we need to resolve the most precise type for each variable on the fly to detect type conflicts as early as possible. Therefore, we need to take an *eager* approach in identifying variable type. That is, whenever a more precise type is resolved, this new type information will be immediately propagated to the other variables sharing the same type, so that all of these variables have the most precise type at this point of execution.

Based on the adapted algorithm, the data type scope tracks the data type information of memory locations and registers. It generates alerts when other type of data are misused as pointers, or when data pointers are misused as control pointers.

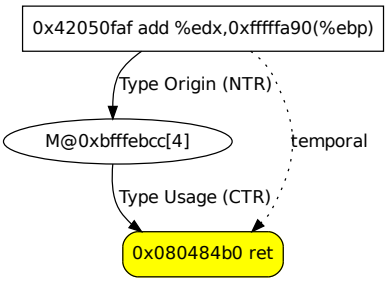

Figure 4. The key-step graph of a simple format string exploit.

3.3 Generating key-step graphs

After the data type scope identifies pointer misuses, the diagnosis engine extracts the causal dependencies for these conflicts to reveal the key steps of the given exploit. To assist security analysts, we represent these dependency relationships using a graph. This *key-step graph* consists of a set of vertices V and a set of edges E.

- **Set of vertices V.** A vertex is either an instruction or a variable involved in a pointer misuse. Each conflict will be represented in at least three vertices: a vertex for the variable with a pointer misuse, a vertex for the instruction that defines the original type of this variable, and a vertex for the instruction that causes type conflicts.

- **Set of edges E.** An edge represents the relationship among vertices. We use edges to record type relationship. There are three kinds of edges. "Type Origin" edge is from the instruction that defines or resolves the current type of a variable to the variable vertex that has a conflict type. "Type Usage" edge originates from the variable with pointer misuse to the instruction that causes the pointer misuse when using this variable. "Temporal Order" edge manifests the temporal relations between the instruction vertices.

Figure 4 shows the key-step graph for the following simple format string exploit, where an integer counter in the `printf` instruction is used to overwrite the return address.

```
printf("%p%p%p%n");
```

To better illustrate the attack, we use different shapes to represent different types of vertices. The ellipse vertices are variables involved in the pointer misuses; the shaded rectangles with round corner are the instructions where pointer misuses are detected; the rectangles are the instructions that define the existing type of the variables. The labels of the variable vertices indicate the variable's type (M@ for memory and R@ for register), the variable's unique identifier (memory address or register name), as well as the variable's size measured in bytes (denoted as a number in the square brackets after the variable's unique identifier). The labels of the instruction vertices indicate the instruction's address, disassembled instruction. It can also include the optional operand information, as well as the function and library the instruction belongs to.

In this example, the pointer misuse is generated on the `ret` instruction at the instruction address *0x080484b0*. It uses variable at the address *0xbfffebcc* as a control pointer (return address), but the variable is inferred as an integer from an `add` instruction at the address *0x42050faf*. This attack is a tradition attack without sophisticated attack steps, so there is only a single key step in the attack.

Usage of the key-step graph. Note that this key-step graph illustrates the big picture of an exploit, but it does not convey the complete knowledge of the exploit. One usage of the key-step graph is to show the similarity among attacks. For example, all attack techniques based on the SEH mechanism will have similar key-step graph. Alternatively, the diagnosis result by PointerScope can be further enhanced by dependency-based diagnosis. For example, using the standard slicing approach [5], we can extract instructions related to one attack step for investigators to investigate, including all the instructions that create and propagate the misused variables.

4 Implementation

In this section, we describe the implementation of PointerScope in general, and discuss a few challenges we faced in implementation.

4.1 Implementation details

We prototyped PointerScope on the Linux system. The execution monitor component is based on the TEMU [35, 39] component of BitBlaze [1, 34], which extends the QEMU [8, 29] whole system emulator to generate instruction traces for specified programs. Basing PointerScope on TEMU has three advantages for vulnerability diagnosis. First, PointerScope can be easily extended to support a new guest OS. Currently, it supports Windows 2000, Windows XP, and multiple Linux versions. Second, this emulation environment makes it easy to set up different configurations and software packages to test exploits. Third, it also provides a good isolation between the exploit testing environment within virtual machine and the analysis environment on the host. Hence, even though the exploit may have

compromised the testing environment, the host machine remains intact. For each instruction executed by the program, our execution monitor records the raw instruction and the content of all its operands, including the implicit operands. The data type scope and diagnosis engine are implemented as standalone utilities, in about 3.6K line of C code.

4.2 Challenges in inferring types

Information from instruction addressing. The first challenge is caused by instruction addressing. The x86 instruction set supports various addressing modes. Each addressing mode has its typical pattern, which reveals information about the typical type of instruction operands. However, the operands may be used differently by the compiler. In particular, the x86 instruction set supports the addressing mode *base-index with displacement*, where a memory address is composed in the format of `base + index + displacement`. For example, the instruction `movl $0x8(%eax, %ebx), %ecx` stores the memory content of `%eax + %ebx + $0x8` into the register `ecx`. Typically, the first register `eax` is the base, which is a pointer; The second register `ebx` is the index, which is an integer. However, the compiler may exchange the base register with the index register or even with the displacement, for example, `$0x8(%ebx, %eax)`, as both instructions produce the same result. We need to decide which register represents the base.

We observed that, in most cases, the resulting memory address is close to the value in the base register. Therefore, we use the value of the registers as a hint and treat the register whose value is closest to the memory operand's address as the base register, which is a pointer.

Special usage of `lea`. Another challenge is from the special usage of instructions. A typical example is `lea`. It is designed to load effective addresses, but it is often used as an optimized way to carry out numerical calculation. For example, `lea $0x8(%eax, %ebx), %ecx` computes `%eax+%ebx+$0x8` and puts the result into the register `ecx`. So we should treat the `lea` instruction as a numerical calculation, without resolving the computation result as an address. Another example is the instructions `shl` and `shr`. They are considered as multiplication and division operations in our implementation. These two instructions are originally for bit operations, but they are widely used for multiplication and division.

Equivalent instruction sequences. A third challenge is from equivalent instruction sequences:

```
not     %ebp
or      $0x3,%ebp
not     %ebp
```

In the above real-world example, the instruction sequence produces the same result as a single instruction `and $0xfffffffc,%ebp`, which is used to align a pointer by clearing its lowest two bits. However, after applying the `not` operation twice on `ebp`, which has a pointer type, the type information is lost under our common type rules. Looking at individual instructions separately will not catch the instructions' intention, and thus make wrong type assignments. Similarly, the `sub` and `neg` instructions can have the similar effects on pointer as `not`.

In this implementation, we solve this problem by recognizing the common patterns and treat them as special cases.

Handling memory copy operations. Programs often copy a memory region from one variable to another area in a byte-by-byte fashion. In PointerScope, every write operation creates a new variable. So truncation and combination will happen to the new copy of variables in the destination region and the original type information is lost.

To solve this problem, we keep a dependency reference for every byte inside a variable. For example, before using a variable A, we will check the dependency of every byte in A. If all bytes in A depend on another variable B, PointerScope assigns the type of B to A first. In this way, we can recover variable information from memory copy operations.

5 Evaluation

We evaluated our approach using a few real-world exploits on both the Windows and the Linux platform. For sample collection, we selected recent non-trivial memory-corruption attacks that have exploits available in the MetaSploit [24] framework. Not surprisingly, most of the attacks are drive-by download attacks. Such attacks commonly rely on sophisticated exploit techniques to bypass the existing defense mechanisms, and malicious web pages often go through heavy transformations in the browser, which bring significant challenge to the traditional dependency-based analysis techniques. Thus we believe PointerScope's result on such attack will demonstrate the potential of our technique. The vulnerabilities and exploits are summarized as follows.

- CVE-2010-0249: Microsoft IE HTML Object Memory Corruption. Windows platform.
- CVE-2009-3672: Microsoft IE "Style" Object Remote Code Execution. Windows platform.
- CVE-2009-0075: Microsoft IE Document Object Handling Memory Corruption. Windows platform.
- CVE-2006-1016: Microsoft IE Javascript IsComponentInstalled Overflow. Windows platform.

CVE	Attack Technique	Runtime	Conflicts	Trace Size	Slice Size
CVE-2010-0249	Uninitialized memory; heap spray	18m23s, 8m30s	11	307,987,560	48,404,242
CVE-2009-3672	Incorrect variable initialization; heap-spray	3m10s, 31s	2	22,759,299	955,325
CVE-2009-0075	Uninitialized memory; heap spray	25m, 21m16s	6	411,323,083	44,792,770
CVE-2006-0295	Heap overflow; heap spray	3m5s, 1s	3	808,392	34,883
CVE-2006-1016	Stack overflow; SEH exploit	4m59s, 1m33s	3	64,355,691	1,334,253
CVE-2006-4777	Integer overflow; heap spray	1m45s, 40s	3	2,632,241	1,669,751
CVE-2006-1359	Incorrect variable initialization; heap spray	11m58s, 13s	2	8,336,193	29,520
CVE-2010-3333	Stack overflow vulnerability; SEH exploit	18m53s, 7m24s	1	236,331,307	814,305
CVE-2010-3962	Incorrect variable initialization; heap spray	10m36, 15s	2	9,281,019	78,704

Table 2. Summary of evaluation. The "Runtime" shows two pieces: the first is the time spent on generating the execution trace and the second is the time spent on generating the key-step graph from the trace. The "Conflict" lists the number of pointer misuses detected. "Trace size" is the total number of instructions in the execution trace. "Slice size" is the number of instructions in the slice that the exploit is dependent on.

- CVE-2006-0295: Firefox location.QueryInterface code execution. Linux platform.
- CVE-2006-4777: Microsoft IE Daxctle.OCX KeyFrame Method Overflow. Windows platform.
- CVE-2006-1359: Microsoft IE createTextRange Code Execution. Windows platform.
- CVE-2010-3333: Microsoft Word RTF pFragments Stack Buffer Overflow. Windows platform.
- CVE-2010-3962: Microsoft IE CSS SetUserClip Memory Corruption. Windows platform.

5.1 Summary of effectiveness and performance

The result of our evaluation is summarized in Table 2. The execution traces' sizes range from about 30K instructions to 411M instructions. Accordingly, it took PointerScope from 1.26 second to about 27 minutes to diagnose the exploits. In each diagnosis, a significant portion (more than 60%) of time was spent on generating the trace.

PointerScope detected all control hijacking behaviors through pointer misuse detection, and correctly characterized the attacks' key steps. We will analyze the details of selected exploits later in this section.

In addition, we collected the following statistics on how PointerScope can improve the efficiency of attack analysis by comparing with dynamic program slicing technique [5] and dynamic taint analysis [25]. Starting from the instruction i that transfers control into malicious payload, we use program slicing on the execution trace to extract all the instructions that influence the instruction i via direct and indirect data dependency. The number of instructions in this slice is listed in the column "Slice Size". As a reference, we also list the total number of instructions in the execution trace for each exploit sample. We can see that although dynamic program slicing can generally extract a small slice relative to the total trace size, the sheer size of a slice is still huge (from tens of thousands to several million instructions).

As for dynamic taint analysis, we mark the exploit input as tainted and track the tainted input propagating through data dependency. Then an alarm is triggered whenever the instruction pointer becomes tainted. We investigated two taint propagation policies: `P1:` direct data dependency only; and `P2:` indirect data dependency included. We made the following observations. On one hand, for all of these exploits except CVE-2006-0295, no alarm is raised if only direct data dependency is tracked. This is because these exploit inputs propagate through indirect data flows. On the other hand, tracking indirect data dependency induced too many alarms, ranging from about 1K to 759K. This phenomenon coincides with Slowinska and Herbert's finding [32]. Therefore we can see that dynamic taint analysis becomes less effective for detecting and analyzing these sophisticated memory-corruption attacks. In contrast, PointerScope detected a small number of pointer misuses, which pinpoint the key attack steps.

5.2 Case studies

We use several representative samples to describe the detailed analysis results for them.

5.2.1 Microsoft IE Javascript IsComponentInstalled overflow

This attack is similar to the attack described in Section 2. It exploits a stack overflow vulnerability to overwrite the

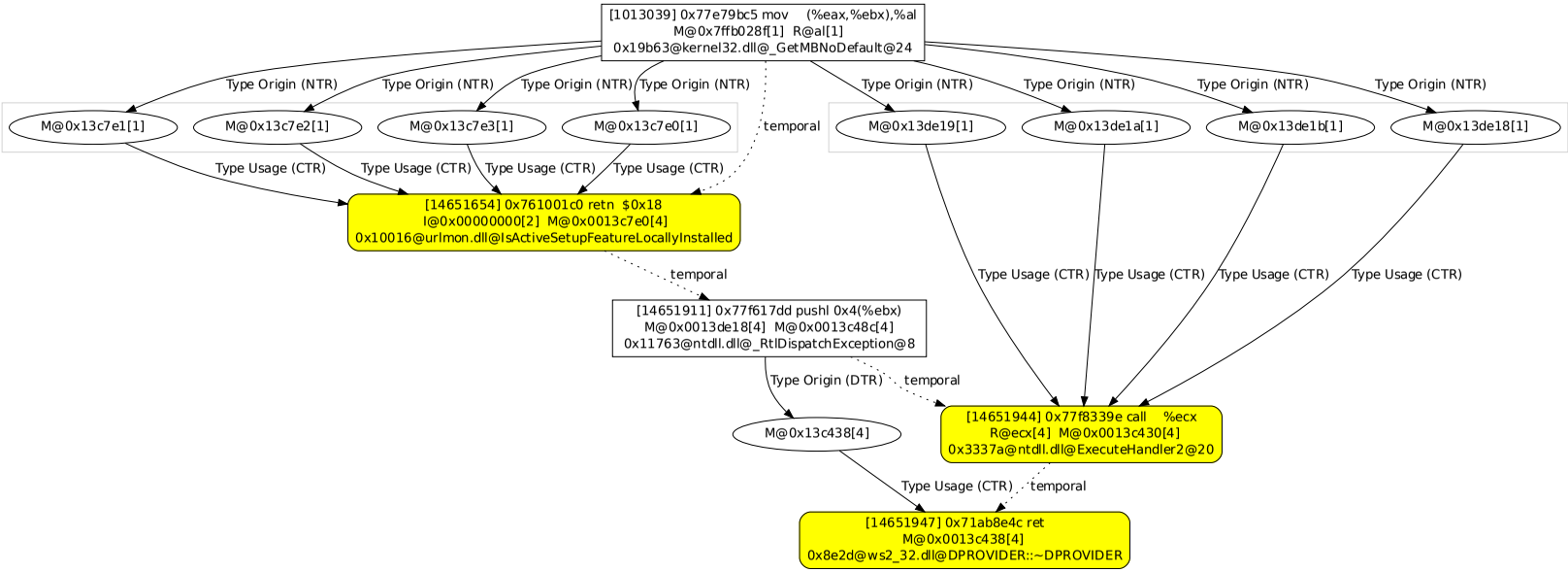

Figure 5. Key-step graph for IE Javascript IsComponentInstalled overflow.

stack, including the return address and the exception registration record. The overwritten return address triggers an exception. When preparing to call the exception handler, Windows pushes the exception registration record's address r on the stack as one of the arguments to exception handler. This is a key step that enables the attack to navigate to the injected code. Finally, the overwritten exception handler leads to a `ret` instruction that uses the address r, and activates the injected code.

The attack graph is shown in Figure 5. It includes three conflicts, where two variables of data type are used as control pointers and a data pointer is used as control pointer. From the graph, we can see that the conflict-causing variable, the return address, is pushed onto the stack as a data pointer in the exception handling function `_RtlDispatchExcpetion`. Moreover, the `ret` instruction was reached by a `call` instruction from the function `ExecuteHandler2` in the library `ntdll.dll`, which causes a pointer misuse when using data as control pointers. This variable used by the `call` is the overwritten exception handler pointer on the stack. Therefore, with help of PointerScope, we correctly capture the key steps for this SEH-based attack.

Once the key steps of the attack are identified, we employ data dependency analysis on each step to help us understand the attack mechanism. The first pointer misuse is on the instruction `retn $0x18`, it is the return instruction in function `IsActiveSetupFeatureLocallyInstalled`. But by following data dependency, we can see that the return address is combined by the bytes which have been copied by a `mov` instruction, not from a `call` instruction. This is the typical scenario of stack overflow. For the second pointer misuse is on the instruction `call %ecx` from function `ExecuteHandler2`. This call instruction, similar with `retn $0x18`, use a variable combined by bytes copied by instruction `mov (%eax,%ebx),%al`. And if we check the assembly code of function `ExecuteHandler2`, we know the call target is an exception handler. So this pointer misuse is caused by overwritten exception handler. Another interesting fact is that both pointer misuses are caused by a single overflow. The last pointer misuse is on a simple return instruction. Its return address is *0x0013de14*. There was a `pushl 0x4(%ebx)` from function `_RtlDispatchException`, which pushed a variable at *0x0013de18* onto the top of stack. So we know *0x0013de18* is a data pointer. Since `%ebx + 0x4` is data pointer, it is easy to understand that `%ebx + 0x0`, whose value is *0x0013de14*, is also data pointer. This is misuse from data pointer to control pointer. By checking the assembly code and definition of function `ExecuteHandler2`, we understand that the variable pushed onto stack is the parameter `EstablisherFrame` for exception handler.

Putting every piece together, we get a complete story of the attack, which employed a buffer overflow to overwrite the return address of `IsActiveSetupFeatureLocallyInstalled` and the SEH structure on the bottom of the stack. It used a invalid value to trigger an exception and hijacked control flow by the overwritten exception handler. Finally, it located shellcode by parameter `EstablisherFrame`, which is prepared by function `ExecuteHandler2`.

We can also look for root cause of the exploit. The vertex `retn $0x18` in function `IsActiveSetupFeatureLocallyInstalled` is the conflict where the corrupted return address is used. The pointer misuse is caused by using a few data bytes as a pointer. After locating instructions that write into these bytes in the trace, we found a big loop that starts to write at the address *0x0013c4ed*. This is at the beginning of the overflowed buffer. It is at the offset 755 of the function's stack frame. Since the corrupted return address is highlighted, we identified the vulnerable function `IsActiveSetupFeatureLocallyInstalled`. Such information helps developers to quickly identify and fix the problem.

5.2.2 Microsoft IE "Style" object remote code execution

This attack is caused by a vulnerability in the class `CDispNode`'s member function `SetExpandedClipRect`. It uses an index value from the HTML page to compute an offset `o` from an array `a`, and find a flag variable located at `o` bytes before the instance of `CDispNode`. It then sets the second bit of the flag (`or $0x2, %eax`). However, the array `a`'s first entry is incorrectly initialized as zero, which makes the offset `o` zero when the index value is zero. Therefore, the address of the `CDispNode` instance is taken as the address of the flag. Now the flag is actually the pointer to the instance's virtual function table. When the function ORs the flag with *0x2*, the pointer to the virtual function table is incremented by *2*. As a result, the pointer to the first virtual function is taken from an address off by *2*, which happen to be an address pointing to the heap. A call to the first virtual function will execute code attackers prepared on the heap.

As Figure 6 illustrates, there are two pointer misuses. The instruction at the bottom is the step where the control transfers to attacker's code. This is an interesting case in which both the operand content and operand address have conflicts. Following the operand content conflict, we found this instruction used parts of two other pointers as the target for `call`. This is clearly wrong. To find the reason, we followed the conflict in the operand address, where we found the address (data pointer) is from the output of an `or` instruction (integer).

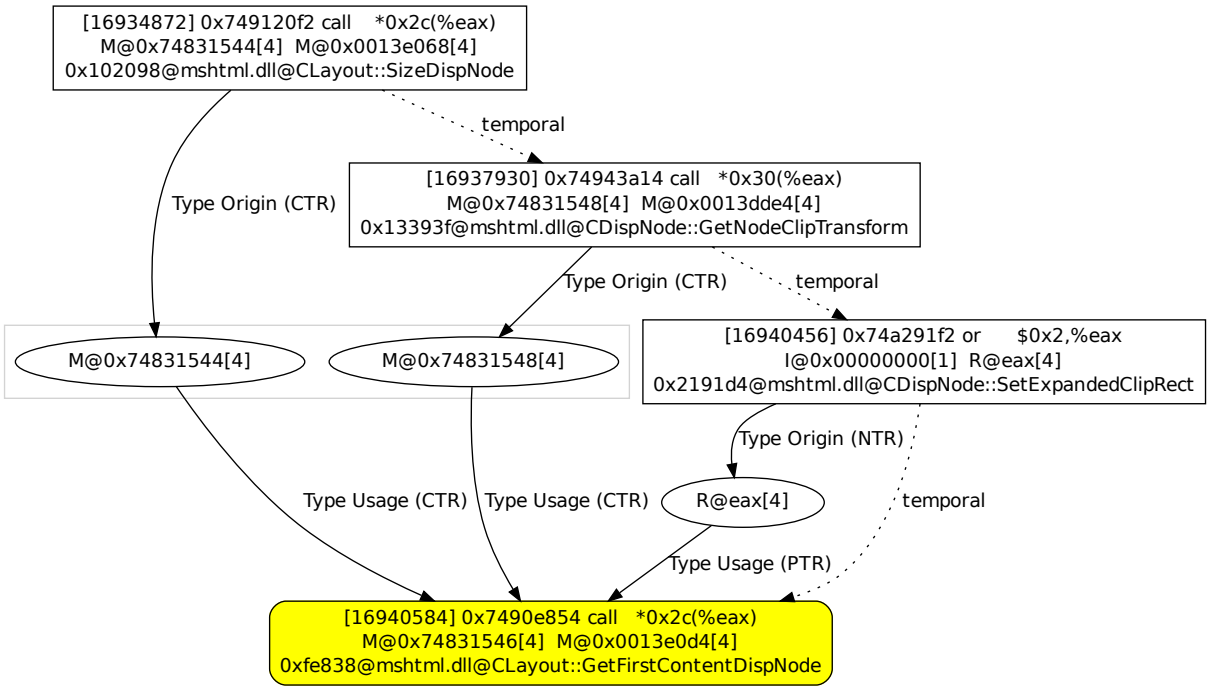

Figure 6. Key-step graph for Microsoft IE "Style" Object Remote code execution vulnerability.

After the key steps of the attack are identified, we use data dependency analysis and slicing on each step to help us understand the attack mechanism. This call instruction contains two misuses. The first one is on the call target. From data dependency of call target we know that this target actually is combined by two parts. Each of those two parts came from a variable who is used to be call target. Then let us look at the address of the highlighted call target. It is computed like `%eax + 0x2c`. This is a typical style of virtual function call and `%eax` is supposed to be the base of virtual function table. And it is easy to find out from data dependency that the three `eax` that are used by non-highlighted instruction `call *0x2c(%eax)`, `call *0x30(%eax)` and `or $0x2,%eax` is the same variable. But this variable is change by instruction `or $0x2,%eax` in Figure 6, which is not supposed to happen.

After we understand the malicious control transfer is triggered by the `or` instruction, we performed backward slicing on its operands, and found the root cause of this exploit nine instructions away. It is an instruction at the address *0x74a291da*, which uses the incorrectly initialized first entry of `a`. Therefore, PointerScope significantly reduces the amount of analysis needed in identifying the root cause of this exploit.

5.2.3 Microsoft IE object tag buffer overflow

This attack exploits a buffer overflow vulnerability in handling the MIME type of the `Object` HTML tag. The vulnerable code converts each "/" character in the MIME type string into the sequence "_/_" after checking the destination buffer size using the size of the original MIME type. Thus, a MIME type with lots of "/" characters will overflow the destination buffer and overwrite the return address.

PointerScope identified that the injected code started to execute after a `ret` instruction, whose return address is created by a `push` instruction that pushes the `esp` register on the stack. The `push` instruction was preceded by another instruction causing pointer misuse. It is the `ret` instruction that uses the overflowed return address. The above three instructions are executed in a sequence. Thus, this attack puts its shell code right after the overflowed return address, and "returns" to an instruction sequence `push %esp; ret`. By understanding these key steps, we understand that this is another mechanism that redirects the execution into the shell code on the stack when the stack location cannot be predicted.

Program	False Positive	Trace Size
AcroRd32.exe	3	328,429,372
calc.exe	0	51,573,864
cmd.exe	0	120,69,909
firefox.exe (www.google.com)	3	277,982,830
iexplore.exe (about:blank)	1	119,996,863
iexplore.exe (www.google.com)	2	317,299,855
msmsg.exe	0	64,033,207
mspaint.exe	0	80,211,147
notepad.exe	0	66,073,327
Skype.exe	5	205,427,083
winword.exe	5	194,218,192

Table 3. Summary of false positive evaluation. The "False Positive" lists the number of distinct false positive. "Trace size" is the total number of instructions in the execution trace.

5.3 False Positive Analysis

Benign program execution may also trigger pointer misuses for performance optimization or other reasons. Since we aim for offline security analysis, having a small number of false positives is not a big concern. Furthermore, we can use a white-list to filter out the known false positives from the final diagnosis report.

We conducted experiments for false positive evaluation using a collection of Windows utilities, such as Notepad, Internet Explorer, etc. We used those utilities to perform their typical operations, such as surfing common web portals and edit documents. We took traces of these operations and analyzed the conflicts. The result is shown in Table 3. We summarized the common patterns causing conflicts into the special cases used in our type operation described in Section 4 and Appendix.

We encountered some interesting false positives in our experiment. These false positives cannot be handled by PointerScope's type operation algorithm.

The first false positive comes from the function `___sbh_alloc_block_from_page` in `mshtml.dll`.

```
mov   0x8(%ebp),%ecx
mov   (%ecx),%edi
mov   %edi,0x8(%ebp)
lea   0xf8(%ecx),%esi
add   %edx,(%ecx)
sub   %edx,0x4(%ecx)
imul  $0xf,%ecx,%ecx
lea   0x8(%edi),%eax
shl   $0x4,%eax
sub   %ecx,%eax
```

Suppose `%ecx` is `p1` and `%edi` is `p2`. The above instructions actually compute `(p2 + 8)*16 - p1*15`, which is `p1 + (p2 - p1 + 8)*16`. According to the instructions before this code segment, the `ecx` contains a memory address that is aligned to page boundaries, which should be a pointer to an empty page. And `p2` is the first member of a structure at the beginning of the empty page. So the effect of this computation is to generate a pointer pointing to another structured data in the page, as indicated by `p1 + (p2 - p1 + 8)*16`. However, the original computation `(p2 + 8)*16 - p1*15` seems meaningless. We believe this is the result of compiler optimization.

The tracing on Microsoft Word shows a complex encoding and decoding process. From the report, we can derive that it first encodes an internal variables using a series of bit operations byte by byte, and then decodes it when using those variables later. Here is a piece of code from `winword.exe` trace file.

```
mov (%edi),%cl
mov 0x27(%esp),%ebx
and $0xff,%ecx
and $0xff,%ebx
xor %ebx,%ecx
xor $0x100,%ecx
shr $0x1,%ecx
and $0xff,%eax
shr $0x1,%eax
xor %eax,%ecx
sar $0x1,%ecx
```

In this code slice, variables are treated as raw data. We cannot restore the original type information inside this structure due to the complex encoding and decoding operation. So we have to filter this kinds of false positives by white-listing.

For Skype, most of false positives come from one special case when it takes a fixed value directly as a pointer, which is calculated from an immediate through a fixed series of operations. Actually, the `call` instruction is using an immediate as target address. Our data type scope will recognize this scenario as a conflict of using *INT* as *PTR*. We need handle this through a special pattern to recognize using fixed value as a pointer by tracking operations on immediate. Here is a code slice from Skype.

```
mov 0x29de027f,%eax
ror 0x10, %eax
add 0x47a7bc4,%eax
neg %eax
add 0x75a1292,%eax
call %eax
```

6 Limitations and Future Work

In this section, we discuss the limitations of our current implementation and future work.

Integer overflows The current type system of PointerScope cannot be used to diagnose integer overflow attacks, which is caused by the confusion between signed and unsigned integers. We may extend the type system to distinguish signed and unsigned integers. However, this extend type system may introduce lots of innocent type conflicts that appear frequently in normal program execution. For instance, conflicts between signed and unsigned integers can be commonplace in normal programs. Nevertheless, it would still be an interesting research problem how to eliminate innocent type conflicts in an extended type system. We plan to investigate this problem in the follow-up work.

Self-modifying Code We noticed that for some normal program executions, PointerScope falsely detected several type conflicts. These conflicts came from self-modifying code. A variable is firstly used as a data pointer to write into a code region, and then used as a control pointer to jump into the code region. It causes a type conflict between *DTR* and *CTR*. This is not an uncommon case, especially in Windows. To allow self-generating code, we can use a learning-based approach to identify the normal code-generation locations, and use a white-list of these locations to avoid reporting these innocent type conflicts.

Dependency among key steps PointerScope identifies the key steps, but does not analyze the dependencies among the steps. PointerScope may also miss key steps that do not incur type conflict. As the next step, we propose to combine PointerScope with dependency-based analysis, such as taint analysis, to perform more complete diagnosis to memory-corruption attacks.

Although the key-step graph highlights important steps of an exploit, manual efforts are still needed to analyze the exploit comprehensively. In the follow-up work, we will also investigate in automated solutions that can further reduce manual efforts.

7 Related Work

Attack Diagnosis Techniques. Some existing diagnosis techniques [12, 18, 20, 25, 28, 38, 40] automatically analyze the attack and illustrate how this attack has happened.

BackTracker [18] builds up a dependency graph between OS objects, such as processes, files, sockets, and so on, and this dependency graph can be used to trace back the origin of an intrusion. In comparison, exploit diagnosis needs to reason about dependency between instructions, so a dependency graph in the OS object level is too coarse grained.

Dynamic taint analysis [25] keeps track of the data dependency originated from untrusted user input at instruction level, and detects an exploit on a dangerous use of a tainted input. Then a data dependency graph can be constructed from the detection point. However, as a study shows, when tracking indirect data dependency is blindly enabled, the amount of tainted data and dependency information will explode, generating tremendous amount of false alarms and irrelevant dependency relationships. Moreover, taint analysis may miss important attack steps because these steps depend on the tainted input via control flow. DTA++ [17] enhances traditional taint analysis by handling a targeted subset of control flow. Based on dependency analysis, Argos [28] detects zero-day attacks and generates network-level signatures to prevent future attacks, while Vigilante [12] generates host-level filters. PointerScope and taint analysis have complementary advantages: PointerScope highlights important instructions among a huge amount of dependencies, while taint analysis focuses on such instructions to generate detailed diagnosis.

Several solutions [19, 20, 38, 40] perform dependency analysis to analyze a piece of malware and extract valuable insights about its attack mechanism. In particular, Panorama [40] and HookFinder [38] also generate dependency graphs to facilitate security analysts.

Two exploit diagnosis systems take memory analysis approach [22, 37]. When an exploit attack is detected, these tools capture the memory snapshot on this moment, and then try to find evidence on this memory snapshot to reason about the root cause of this attack. It is infeasible for this static analysis approach to understand the entire process of an exploit attack.

Defense and evasion techniques in memory-error exploits. There are a wide-range of solutions to detect and prevent software exploits. Techniques, such as control-flow integrity (CFI) [4], data-flow integrity (DFI) [10], and WIT [6] proactively defeat exploits by enforcing program integrity models, which are derived through extensive analysis of program and often need program source code. In contrast, PointerScope aims to diagnose exploits by inferring types from binary execution, which can complement the above solutions when they are not applicable. Defense mechanisms like StackGuard [36], PointGuard [13], ASLR (i.e., address space layout randomization) [3, 9], and DEP (i.e., data execution prevention) [2] are implemented and deployed in software and the operating system to defeat software exploits. They break some of the key steps of an exploit, while the goal of PointerScope is to get the complete picture of exploits for diagnosis.

The attackers always invent new evasion techniques to bypass these defense mechanisms. For instance, to bypass DEP, a returned oriented malicious payload reuses the existing code in vulnerable program and runtime libraries [31]. More specifically, small code snippets ending with a return instruction are stitched together to implement Turing-complete malicious functionalities. More recently, researchers showed that return-to-libc attacks can be accomplished without using return instructions [11]. Moreover, bypassing both DEP and ASLR is possible, as a study shows [16]. Recent news about enhanced Aurora exploit confirms this statement [7].

As this arms race continues to escalate, the objective of our work is to automatically capture the characteristics of a new exploit and understand the weakness of the existing defense system, so that the defense system can be enhanced promptly to thwart similar exploit techniques.

Type and data structure recovery from binaries Recovering the type information from binary programs is valuable for computer security. ASI [30] proposes an approach to identify aggregate data structures. REWARDS [23] takes a dynamic approach to type inference for binary executables. It mainly targets memory forensics, and thus its goal is to recover as much abstract type information as possible. When a variable may hold multiple types, it chooses to keep them all, to maximize the abstract type knowledge. TIE [21] takes a static approach to type inference. It infers type for registers and memory variables simply from a binary executable. Laika [14] uses Bayesian unsupervised learning to figure out data structures of binaries, and used them to detect common polymorphic botnets. Howard [33] identifies data structures more accurately through extensive dynamic analysis.

All solutions in this category aim to extract complete information from binary programs to assist understanding and reverse engineering of binaries. In contrast, PointerScope aims to detect pointer misuses for exploit diagnosis, so it needs to accurately resolve the basic type for each variable. It also uses a much simpler type system to guide the diagnosis of complex attacks. For example, after analyzing the entire execution, REWARDS may report that the type of a union field is a set of types: pointer and integer. For the same situation, PointerScope determines the exact type of this union field on each related instruction to detect conflicts. Increasing the granularity of types of PointerScope may help to extend the key-step identification technique beyond memory-corruption exploits, but it will increase the number of false positives in type conflicts.

8 Conclusion

In this paper, we presented the design and implementation of an exploit diagnosis tool, called PointerScope. The goal was to analyze sophisticated software exploits, which pose substantial challenges to the existing analysis techniques because of the complexity of web browsers and extensions and sophistication of exploits. To tackle this challenging problem, we leveraged a key insight that the key steps of these memory corruption attacks are often pointer misuses. To catch these pointer misuses, we designed a small type system on x86 instructions and devised a type inference algorithm to resolve type information during the binary execution and detect type conflicts. Then we performed dependency analysis on these type conflicts to understand the inner-working of an exploit. We evaluated multiple recent sophisticated memory-corruption exploits, and demonstrated the capability and effectiveness of of PointerScope.

Acknowledgments

We thank the anonymous reviewers for their insightful comments for improving this paper. This work is supported in part by an NUS Young Investigator Award R-252-000-378-101, and the US National Science Foundation (NSF) under Grants #1018217 and #1054605. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] The bitblaze project. `http://bitblaze.cs.berkeley.edu`.

[2] Data execution prevention. `http://technet.microsoft.com/en-us/library/cc738483(WS.10).aspx`.

[3] The pax team. http://pax.grsecurity.net.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 340–353, 2005.

[5] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland'08)*, pages 263–277, 2008.

[7] Aurora Exploit Retooled To Bypass Internet Explorer's DEP Security. `http://http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=222301436`.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 11–11, 2006.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of ACM Conference on Computer and Communications Security (CCS'10)*, Oct. 2010.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'05)*, 2005.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, 2003.

[14] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 255–266, 2008.

[15] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1982.

[16] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, Honolulu, Hawaii, USA. IEEE.

[17] M. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[18] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 223–236, October 2003.

[19] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, July 2006.

[20] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, February 2009.

[21] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, February 2011.

[22] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[23] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, February 2010.

[24] Metasploit penetration testing framework. `http://www.metasploit.com`.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[26] M. Nicholls. Tutorial: Seh based exploits and the development process. `http://www.ethicalhacker.net/content/view/309/2/`.

[27] M. Pietrek. A crash course on the depths of win32^{TM} structured exception handling, 1997. `http://www.microsoft.com/msj/0197/exception/exception.aspx`.

[28] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of the 1st ACM SIGOPS/EuroSys European conference on Computer systems (EuroSys'06)*, April 2006.

[29] Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[30] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, pages 119–132, 1999.

[31] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conferences on Computer and Communication Security(CCS'07)*, pages 552–61. ACM Press, Oct. 2007.

[32] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)*, April 2009.

[33] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*. The Internet Society, 2011.

[34] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of International Conference on Information Systems Security*, Hyderabad, India, 2008. Keynote invited paper.

[35] TEMU: The BitBlaze dynamic analysis component. `http://bitblaze.cs.berkeley.edu/temu.html`.

[36] P. Wagle and C. Cowan. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255, 2003.

[37] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[38] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[39] H. Yin and D. Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[40] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th*

ACM Conferences on Computer and Communication Security (CCS'07), October 2007.

A Special usage of instructions

In this section, we summarize the special usage of instructions we identified and encountered during evaluation.

A.1 Special use of bit instructions

Special usage of `and`

- zero test: `and` is used to test whether a variable is zero or not.

  ```
  and %eax, %eax
  ```

- mask usage: `and` can be used to align memory address by masking out lower bits.
- abnormal usage: We observed two unusual scenarios used by `and`.

  ```
  and 0xfffffffb, %eax
  and 0x03, %eax
  ```

  In the first scenario, assuming that the `eax` register is of type *PTR*, we should keep it as the *PTR* type after `and` operation. In the second scenario, no matter what type the register `eax` is, it will be changed to the type *INT*.

- assignment:

  ```
  and 0x0, %eax
  ```

  In above example, `and` is used to set the value of `eax`. So we create a rule that if the result of the `and` is equal to one of the source variables, the result will inherit the type of that source variable.

Special usage of `or`

- zero test: `or` can be used to do zero test.

  ```
  or $0x0, %eax
  ```

- mask usage: `or` is used to mark a pointer, used by Firefox.

  ```
  or $0x01, %ecx
  ```

- assignment:

  ```
  or $0xffffffff, %eax
  ```

  Similer with `and`, if the result of the `or` is equal to one of the source variables, the result will inherit the type of that source variable.

Special usage of `xor`

- zero set: It can be used to set a variable to zero.

  ```
  xor %eax, %eax
  ```

- encode data: Sometimes `xor` is used to encode/decode data with an integer key.

  ```
  xor $KEY, %ebx ...  xor $KEY, %ebx
  ```

Special usage of `shl`/`shr`

`shl` is considered a multiply operation, `shr` is considered a divide operation. Those two instructions are originally for bit operations, but they are widely used for multiplication and division.

For example, mostly the `memcpy` function takes a source pointer, destination pointer and a byte counter as parameters. The compiler can put the two pointers into `esi` and `edi` and the byte counter into `ecx`, then run `movsb` directly. But for higher efficiency, compiler always uses a two-bit right shift operation to the counter before putting it into `ecx` and uses `rep movsd` instead of `rep movsb`. Similarly, `shl` will be used when the compiler tries to convert a double-word counter into byte counter.

Special usage `setc`

`setc` is used to store a bit of CPU's EFLAG. This is be a boolean variable, but it may be used as an index (*INT*) sometimes.

A.2 Pointer tag in Firefox

```
or 0x01, %ecx
and $0xfffffffc, %ecx
mov %eax, 0x24(%ecx)
```

From our analysis on Firefox, it shows Firefox has a special mechanism on pointers. In Firefox, it uses `jsval` as a pointer. Each `jsval` value encodes the type of data it points to. For example, if lower three bits of a `jsval` value is 0x2, then this `jsval` pointer points to DOUBLE, 0x4 for STRING, and etc. Adding or removing tags from a data pointer is very common operations in Firefox. Tags are added by the `and` instruction and removed by the `or` instruction. `xor` is also used to remove a tag from a pointer. We treat them as special cases.

Instruction Category	Representative Instructions	Type Constraints
Data transfer	MOV, CMOV, PUSH, POP	Type constraint between operands
Control transfer	JMP, JZ, LOOP, CALL, RET	Operands are control pointer
Binary arithmetic	ADD, SUB	Type constraint between operands
	IMUL, MUL, IDIV, DIV	Operands are integer
Decimal arithmetic	DAA, AAA	Operands are integer
Logical	AND, OR	Type constraint between operands
Shift and rotate	SAR, SHR, SAL, ROL, RCL	Operands are integer
Bit and byte	BT, BTS, SETS	Operands are integer
String	MOVS, LODS, STOS	Type constraint between operands
	CMPS, SCAS	None
	REP, REPZ, REPNZ	Type constraint between operands. ECX is integer
IO	IN, OUT, INS, OUT	None
Flag control	LAHF, SAHF, PUSHF, POPF	Operands are integer
	STC, CLC, CMC, STI, CLI	None
Segment register	LDS, LES, LFS, LGS, LSS	None
Misc	MOVBE	Operands are integer
	LEA	Type constraint between operands
	NOP, UD2	None

Table 4. Type constraints of common instructions.