usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

# Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard

Enes Göktaş, *Vrije Universiteit Amsterdam;* Elias Athanasopoulos, *FORTH-ICS;*
Michalis Polychronakis, *Columbia University;* Herbert Bos, *Vrije Universiteit Amsterdam;*
Georgios Portokalidis, *Stevens Institute of Technology*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas

This paper is included in the Proceedings of the
23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

# Size Does Matter
## Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard

Enes Göktaş
*Vrije Universiteit*
*Amsterdam, The Netherlands*

Elias Athanasopoulos
*FORTH-ICS*
*Heraklion, Crete, Greece*

Michalis Polychronakis
*Columbia University*
*New York, NY, USA*

Herbert Bos
*Vrije Universiteit*
*Amsterdam, The Netherlands*

Georgios Portokalidis
*Stevens Institute of Technology*
*Hoboken, NJ, USA*

## Abstract

Code-reuse attacks based on return oriented programming are among the most popular exploitation techniques used by attackers today. Few practical defenses are able to stop such attacks on arbitrary binaries without access to source code. A notable exception are the techniques that employ new hardware, such as Intel's Last Branch Record (LBR) registers, to track all indirect branches and raise an alert when a sensitive system call is reached by means of *too many* indirect branches to *short* gadgets—under the assumption that such gadget chains would be indicative of a ROP attack. In this paper, we evaluate the implications. What is "too many" and how short is "short"? Getting the thresholds wrong has serious consequences. In this paper, we show by means of an attack on Internet Explorer that while current defenses based on these techniques raise the bar for exploitation, they can be bypassed. Conversely, tuning the thresholds to make the defenses more aggressive, may flag legitimate program behavior as an attack. We analyze the problem in detail and show that determining the right values is difficult.

## 1 Introduction

Modern protection mechanisms like data execution protection (DEP) [2], address space layout randomization (ASLR) [26] and stack smashing protection (SSP) [9] are now available on most general-purpose operating systems. As a result, exploitation by injecting and executing shellcode directly in the victim process has become rare. Unfortunately, these defenses are not sufficient to stop more sophisticated attacks.

Nowadays, attackers typically use memory disclosures to find exactly the addresses ASLR is trying to hide [30, 34, 36]. Likewise, there is no shortage of tutorials on how to evade state-of-the-art defenses [14, 28]. Attackers are able to hijack control flow and bypass DEP by reusing code that is already available in the binary itself, or in the libraries linked to it. There are several variations of this exploitation method: return-to-libc [37], return-oriented programming (ROP) [31], jump-oriented programming [3, 6], and sigreturn oriented programming (SROP) [4]. Code reuse attacks, and especially ROP, may be the most popular exploitation method used by attackers today, bypassing all popular defense mechanisms. Even additional and explicit protection against ROP attacks over and beyond DEP, ASLR and SSP, such as provided by Microsoft's Enhanced Mitigation Experience Toolkit (EMET), do not stop the attacks in practice [14].

ROP attacks start when an attacker gains control of the stack and diverts the control to a *gadget*: a short sequence of instructions that performs a small subset of the desired functionality and ends with a `ret` instruction. Since the attackers control the return addresses on the stack, they can make the `ret` of one gadget jump to the start of another gadget, daisy chaining the desired functionality out of a large set of small gadgets.

It is no wonder, then, that the security community has scrambled to find alternative methods to defend software assets. For instance, over the past decade or so, there has been a tremendous amount of research interest in control flow integrity (CFI) [1]—a technique to prevent *any* flow of control not intended by the original program. Unfortunately, CFI is fairly expensive. Moreover, research has shown that attempts to make it faster and more practical by employing looser notions of integrity, make it vulnerable to exploitation again [16].

**KBouncer and friends** Perhaps the main and most practical defense mechanism proposed against ROP attacks nowadays is the one pioneered by kBouncer [25]—grand winner of the Microsoft Blue Hat Prize in 2012. The technique has become quite successful and despite its recent pedigree, it is already used in commercial products like HitmanPro's new Alert 3 service [18].

KBouncer and related approaches like ROPecker [8] use a new set of registers, known as the Last Branch Record (LBR), available in modern Intel CPUs. The registers can be used to log the last *n* indirect branches taken by the program. Using the LBR, kBouncer checks whether the path that lead to a sensitive system call (like `VirtualProtect`) contains "too many" indirect branches to "short" gadgets—which would be indicative of a ROP chain.

The two obvious questions that we need to ask are: what is "too many," and how short is "short"?

Specifically, suppose the defensive mechanism has thresholds $T_C$ and $T_G$, such that it raises an alarm when it sees a chain of $T_C$ or more gadgets of at most $T_G$ instructions each. If the attackers can find just a single gadget greater than $T_G$ that they can simply squeeze in between the others to break the sequence, the defensive mechanism would not detect it. Conversely (and more worryingly), if a program itself exhibits $T_C$ gadgets of at most $T_G$ instructions during normal execution, the defense mechanism would erroneously flag it as an attack.

Implemented carefully, the protection offered by this method is quite powerful, but picking the right values for $T_G$ and $T_C$ is a delicate matter. After all, the former scenario suggests that $T_G$ is too small. However, incrementing $T_G$ may lead to more false positives (FPs) because benign execution paths are more likely to contain $T_C$ such gadgets.

**Contributions** In this paper, we investigate the problem of picking the right values for these two thresholds. We also evaluate whether the solutions proposed today are sufficient to stop exploitation in real software. Specifically, we show that while they raise the bar for exploitation significantly, they can be bypassed. As a demonstration, we discuss a proof of concept exploit against Internet Explorer that bypasses current kBouncer-based defenses. We then analyze the problem by considering the availability of gadgets of different lengths and determining the sequences of gadgets en route to sensitive system calls.

While this work does not fully explore the possibility of FPs with the thresholds used in literature, it shows that defining restrictive thresholds, which do not allow the composition of ROP payloads, is extremely complicated and may not be possible for many applications due to FPs. Finally, we discuss various avenues for ameliorating these techniques and provide evidence that setting the thresholds based on the application at-hand can significantly encumber attackers.

**Outline** The remainder of this paper is organized as follows. Section 2 provides some background information regarding code-reuse attacks and defenses that use gadget-chain length for detection. Section 3 discusses the weaknesses of such approaches, and in Sec. 4 we present the process of creating an exploit that can circumvent them. We propose countermeasures and discuss possible obstacles for their adoption in Sec. 5. In Sec. 6, we present the results of our experiments that indicate that one of the proposed countermeasure can improve detection. Related work is in Sec. 7 and we conclude in Sec. 8.

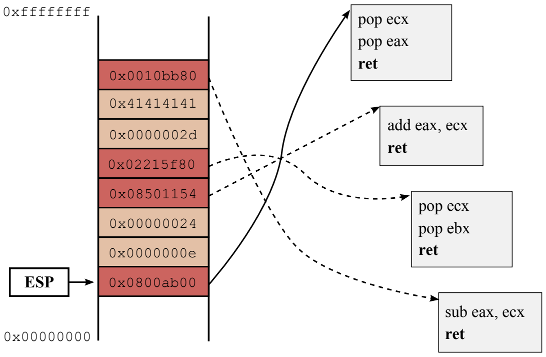


Figure 1: A very simple ROP chain that calculates $0xe + 0x24 - 0x2d$. Result is in the `eax` register.

## 2 Background

### 2.1 ROP and Code-reuse Attacks

ROP attacks are the most common vector for launching code-reuse attacks and require that the attacker gains control of the program's stack. By corrupting the return address of the executing function, upon its return, control is diverted to a *gadget* of the attacker's choosing. Gadgets are small sequences of code that end with a `ret`. By carefully positioning data on the stack, the attacker can make the program jump from one gadget to another, chaining together pieces of already existing code that implement the desirable payload, as shown in Fig. 1. While the gadgets that the attacker chains together are usually short in length (i.e., in number of instructions) and limited in functionality, previous work has shown that the attack is Turing complete [31]. That is, applications contain enough gadgets to perform arbitrary computations.

Creating a working ROP exploit is often a complex, multi-step process. It typically starts with a memory disclosure that allows the attacker to obtain code pointers. Next, the attack may require a variety of further preparations, such as advanced heap feng shui [35] to pave the way for a dangling pointer exploit, stack pivoting, and/or buffer overflows. In addition, the attacker needs to identify useful gadgets and construct a ROP program out of them by setting up the appropriate addresses and

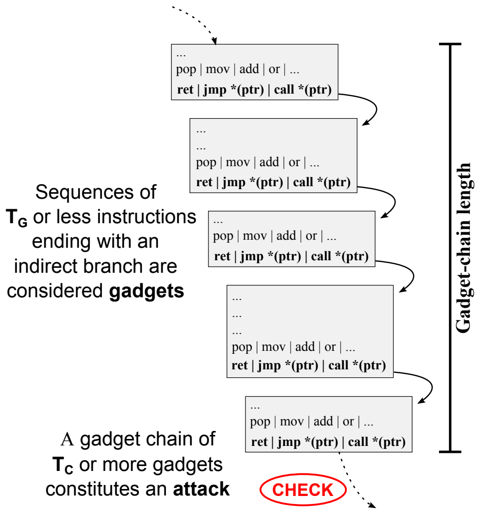


Figure 2: Example of a gadget chaining pattern used to identify code-reuse attacks.

arguments on the (possibly new) stack. Finally, a control flow diversion should start off the ROP chain.

ROP is popular despite its complexity because it provides a way for attackers to bypass defenses like DEP [2]. As a result, many recent works have focused on preventing ROP and other forms of code-reuse attacks [7, 8, 13, 23–25, 31, 33]. Other works have shown that a similar attack can also be performed with gadgets that end with indirect jump or call instructions instead of returns [3, 6, 21].

## 2.2 Monitoring Gadget Chains to Detect Attacks

kBouncer [25] and ROPecker [8] are two of the most easy to deploy solutions to stop ROP-like attacks. They employ a recent feature of Intel CPUs, known as the Last Branch Record (LBR), that logs the last branches taken by a program in a new set of registers [20, Sec. 17.4]. Intel introduced LBR for both the x86 and x86-64 architectures, so that, with the right configuration, the operating system (OS) is able to log the targets of indirect branches (including calls, jumps, and returns) in 16 machine-specific registers (MSR) registers with little overhead. These registers are accessible only from the OS kernel and are continuously overwritten as new branches occur.

A key observation for detecting ROP attacks, in both kBouncer and ROPecker, is that the attacks need to chain together a significant number of small gadgets to perform any useful functionality, like in the example shown in Fig. 2. From a high-level perspective, they include two parameters: the first controls what is the longest sequence of instructions ending with an indirect branch that will be considered a gadget, and the second specifies the number of successively chained gadgets that indicates an attack. We will refer to these two thresholds as $T_G$ and $T_C$. These two parameters control the level of difficulty for performing an attack under these solutions. Increasing $T_G$ or reducing $T_C$ makes the construction of ROP payloads harder. However, overdoing it can lead to false positives (FP), due to legitimate execution paths being misclassified as attacks at run time. In the remainder of this section, we will briefly highlight kBouncer and ROPecker.

### 2.2.1 kBouncer

kBouncer kicks in every time a sensitive API call, like `VirtualProtect()`, `CreateProcess()`, etc., is executed by inserting hooks through the Detours [19] framework for Windows. It then scans the LBR registers to detect if the API call was made by a malicious ROP gadget chain, and terminates the running process if it was.

Two mechanisms are used to determine if there is an attack. The first mechanism aims to identify abnormal function returns. It is based on the observation that ROP chains manipulate control-flow to redirect control to arbitrary points in the program, where the attacker-selected gadgets reside. This constitutes a deviation from legitimate behavior, where returns transfer control to instructions immediately following a call. kBouncer checks the targets of all return instructions in the LBR to ensure that they are preceded by a call instruction. In x86 architectures where unaligned instructions are permissible, this call instruction does not necessarily need to be one actually intended by the program and emitted by the compiler. Any executable byte with the value of *0xE8*, one of the opcodes for the `call` instruction, can be actually considered as an *unintended* call instruction and attackers can use the gadget following it.

Recently, even just using gadgets following *intended* calls was shown to be sufficient to compose ROP payloads [16]. In anticipation of the possibility of such attacks, kBouncer introduced a second mechanism, based on gadget-chain length, to detect and prevent attacks. First, all potential gadgets are identified through offline analysis of an application. Every uninterrupted sequence of at most *20 instructions* ending in an indirect branch is treated as a potential gadget. At run time, kBouncer checks that there is no uninterrupted chain of *eight* or more such gadgets as targets in the LBR. In this case, the maximum gadget length of $T_G = 20$ was selected arbitrarily [25, Sec. 3.2], while through experimentation with a set of Windows applications, it was determined that a safe choice for the gadget-chain length threshold is $T_C = 8$ [25, Sec. 3.2, Fig. 7].

### 2.2.2 ROPecker

Similarly to kBouncer, ROPecker [8] utilizes LBR to detect ROP attacks. However, instead of only checking LBR registers upon entry to sensitive API calls, it introduces a new mechanism for triggering checks more often. It maintains a sliding window of code that is executable, while all other code pages are marked as non-executable. Checks are made each time a permission fault is triggered because control flow is transferred outside the sliding window. The intuition behind this approach is that due to code locality page faults are not triggered very often and ROP attacks are unlikely to use only gadgets contained within the sliding window (between 8 and 16 KB), so a check will be triggered before the attack completes.

Attack detection occurs primarily by checking gadget-chain length, like in kBouncer. However, ROPecker also checks for attacks in future returns by inspecting the return addresses stored in the stack. Potential gadgets are collected offline by statically analyzing applications, but they are defined differently from kBouncer. In particular, a gadget is a sequence of no more than *six instructions* that ends with an indirect branch, but does *not* contain any direct branches. Experiments were conducted with various Linux applications and benchmarks to determine a safe choice for the gadget-chain length threshold that will indicate an attack. The results varied, but a chain of *at least 11* gadgets was determined to be a safe choice. However, using a per-application threshold, if possible, is recommended. To summarize, the maximum length of a gadget is set to $T_G = 6$ and is selected arbitrarily [8, Sec. VII.B], while the safe choice for the gadget-chain length threshold is $T_C = 11$ [8, Sec. VII.A].

In the presence of multiple smaller gadget chains, intentionally created by mixing long and short gadgets to evade the mechanism, ROPecker also proposes accumulating the lengths of the smaller chains across multiple windows and using that instead, to gain a certain degree of tolerance to such attacks. Accumulation is done every three windows, and experimental results showed that an acceptable threshold for cumulative gadget-chain length is $T_{CC} = 14$.

## 3 The Problem

Both systems we study in this paper heavily depend on two parameters, namely $T_C$ (the chain length) and $T_G$ (the gadget length). In this section, we discuss the problem of picking the right values for $T_C$ and $T_G$, and the way attackers can bypass the defenses proposed by kBouncer, ROPecker, and similar approaches.

**What Is the Right Size?** Mechanisms like kBouncer and ROPecker rely on defining gadgets based on the size of instruction sequences ending in indirect branches, and detect attacks based on the size of gadget chains. The problem with such measures is that while they do raise the bar, they are also their own Achilles' heel. By interspersing their ROP code with an occasional longer sequence of instructions (ending with an indirect branch) that will not be registered as a gadget, an attacker can reduce the length of gadget chains, as observed by these systems, and avoid detection.

Figure 3 shows a high-level overview of such an attack. After receiving control through an exploit, an attacker first uses up to $T_C - 1$ *detectable gadgets* (DG). Then, he employs at least one longer *undetectable gadget* (UG), that is, a sequence of more than $T_G$ instructions. To be precise, an attacker may need to use an UG earlier for the first time because a chain of $T_C$ legitimate application gadgets may already exist before he receives control, leading to a longer chain of DGs. If a check is triggered while this chain is still visible in the LBR, the attack will be detected. kBouncer only conducts checks on certain API calls, so an attacker needs to only worry about the number of DGs in the LBR when performing such calls. On the other hand, ROPecker triggers checks more frequently, but, exactly due to this fact, uses less restrictive $T_C$ and $T_G$ parameters. In the worst case, an attacker needs to use an UG first in his ROP chain.

**Weak Control-Flow Enforcement** kBouncer performs an additional check to ensure that the targets of all return instructions in the LBR point to instructions preceded by calls. However, recent work [16] has shown that it is possible to build a ROP payload using such call-preceded (CP) gadgets and evade even stricter control-flow restrictions. As a result, the effectiveness of defenses like kBouncer depends entirely on the $T_C$ and $T_G$ parameters. In the example shown in Fig. 2, the attacker would not be able to link gadgets that are not preceded by calls using function returns.

**Accumulating Gadget-Chain Lengths** ROPecker proposes an extension to tackle exactly the problem of mixing long and short gadgets. They define another parameter, $T_{CC}$, which is the threshold for the cumulative length of gadget chains in three successive windows and, hence, checks. This extension aims to prevent attacks following the pattern shown in Fig. 3. However, ROPecker does not consider instruction sequences including direct branches as gadgets, so an attacker can employ those as alternative shorter UGs. Furthermore, attackers can carefully construct attacks that consist of a small number of gadgets and then inject code, as it was

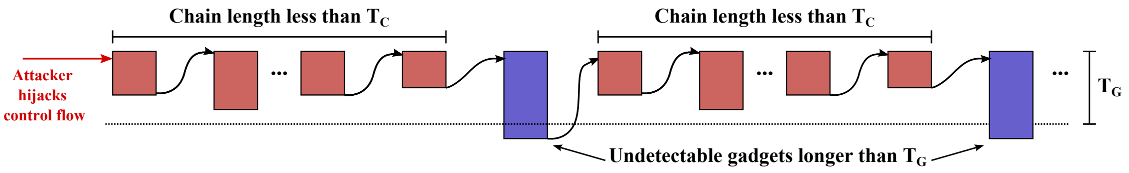

Chain length less than $T_C$
Chain length less than $T_C$
Attacker hijacks control flow
...
$T_G$
Undetectable gadgets longer than $T_G$

Figure 3: Mixing shorter and longer gadgets to avoid detection. Gadgets larger than $T_G$ instructions are not considered as gadgets by both kBouncer and ROPecker. The latter also ignores gadgets that contain direct branch instructions.

done in previous work [16]. Details follow in the next section.

## 4 Proof-of-Concept Attack

In this section, we describe the construction of a proof-of-concept (PoC) exploit, which can compromise a vulnerable binary running under kBouncer. We have selected kBouncer, since we consider it the hardest to evade of the two systems examined in this paper. Recall that kBouncer is based on restricting ret instructions, so that they can only redirect control flow to gadgets preceded by an intended or unintended call instruction, and on a heuristic that scans for long chains of consecutive gadgets, as they are defined by kBouncer. The constructed exploit is generic, because it uses gadgets solely from the `shell32.dll` library which is shared among many widely used applications in Windows, and it is also effective against similar approaches, like ROPecker [8].

Previous work [16] has already shown that it is possible to compose attacks using an even more limited set of gadgets, that is, only gadgets following intended call instructions and starting at function entry points. We build on this prior knowledge to collect the gadgets that are available under kBouncer and show that we can build an exploit that remains undetectable. More importantly, we show that we can construct a very short payload that could not be easily detected unless $T_C$ and $T_G$ are set to considerably more restrictive values.

### 4.1 Preparation

The vulnerability we use to build our exploit is based on a real heap overflow in Internet Explorer [27] and has been also used in multiple other works [16,34] in the past. The first part of the exploit deals with disclosing information to bypass ASLR and then controlling the target address of an indirect jump instruction. Details of the preparation phase can also be found in previous work [16]. Here, we summarize the initial steps that are common with previous work and introduce new actions that are necessary for completing this exploit.

The vulnerability is triggered by accessing the `span` and `width` attributes of an HTML table's column through JavaScript. A great feature of the vulnerability is that it can be triggered repeatedly to achieve different tasks. First, it can be triggered to overwrite the size attribute of a string object, which consequently allows the `substring()` method of the string class to read data beyond the boundary of the string object, as long as we know the relative offset of that data from the string object. The `substring()` method serves as a memory disclosure interface for us. Second, it can be triggered to overwrite the virtual function table (VFT) pointer within a button object. Later, when we access the button object from within carefully prepared JavaScript code, the program will operate on the overwritten data and will eventually grant us control over an indirect jump instruction.

Due to ASLR being in use, we need to exploit the string object to “learn” where `shell32.dll` is loaded at run time, i.e., its base address. Before anything else, we use heap Feng Shui [35] to position the vulnerable buffer, and the string and button objects in the right order, so that we can overflow in the string object without concurrently receiving control of the indirect jump. The following steps are taken to locate `shell32.dll`, the first two steps have been also part of prior work, while the latter was added to achieve our end goal:

1. This vulnerability allows us to easily locate `mshtml.dll`. The button object's VFT contains a pointer to a fixed offset within the DLL. After heap Feng Shui, the button object follows the string object in memory at a fixed distance, so we use the controlled string object to read that pointer and reveal the location of the DLL. `mshtml.dll` contains pointers directly to `shell32.dll`, however, they are located in its *Delayed Import Address Table* (IAT), so they are not available at the time of exploitation.

2. In contrast to `mshtml.dll`, `ieframe.dll` do contain pointers to `shell32.dll` in its normal IAT, which gets loaded during the initiation of libraries. So `ieframe.dll` has the pointers to `shell32.dll` we are looking for available at the time of exploitation. As a result, by learning the base address of `ieframe.dll`, we can achieve our end goal. `mshtml.dll` has pointers

to ieframe.dll available in its Delayed IAT, but to read that, we first need to calculate its relative offset from the string. Since we already know the base address of mshtml.dll, we just need to find out the address of the string object, so we can calculate offsets within the DLL. Fortunately, the button object contains an address that has a constant distance from the beginning of the string object, so by first exfiltrating that, we can calculate the base address of ieframe.dll.

3. Since we now know the base address of ieframe.dll, we exploit the string object once more to read a pointer to shell32.dll, thus revealing its base address. shell32.dll also allows us to locate VirtualProtect() through its own IAT.

Finally, we need to also determine the location of a buffer we control, which we use to store the ROP payload, shellcode, etc. We use heap spraying [11] to create many copies of such a buffer in the process' memory, which has the effect of placing one of the copies at an address that can be reliably determined. Heap spraying is not foolproof, however, it works consistently in this particular case.

## 4.2 Collecting Gadgets

When kBouncer and friends are active the ret instruction can only target gadgets that are preceded by a call instruction. Previous work has referred to such gadgets as call-site (CS) gadgets [16], we will use this term to refer to them. CS gadgets are a subset of all the gadgets available in traditional ROP and JOP attacks, and include gadgets defined by intended and unintended call instructions. So any bytes in the program that could be interpreted as a call instruction, subsequently introduce a CS gadget. Note that the entire set of gadgets is still present in the application, but it can now only be targeted by indirect jump and call instructions. Generally, we will use the CS prefix with gadgets that are call preceded and the type of indirect branch ending the gadget as suffix (e.g., RET or CALL *).

To find usable gadgets, we disassemble the target binary multiple times. We start disassembling from each individual byte in the code segment of each image, until we encounter a stop condition, which can be an indirect control flow transfer, or an invalid or privileged instruction. This is similar to the static analysis phase of kBouncer that determines the locations of gadgets. Like kBouncer, we follow direct branches and calculate the length of a gadget using the shortest number of instructions that can execute from the beginning of the gadget till an indirect branch. This means that in the presence of conditional branches, we follow both paths and use the shortest one as the length of the gadget.

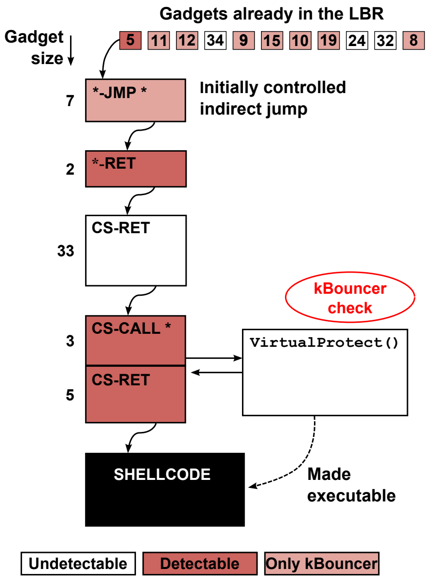


Figure 4: PoC exploit that bypasses both kBouncer and ROPecker. This figure focuses more on the details related with kBouncer, since it uses stricter detection thresholds. Gadgets receiving control through a return are all call-preceded, depicted using the *CS* prefix in the figure. The exploit uses one heuristic breaking gadget to keep the chain of detectable gadgets small and calls VirtualProtect() which triggers a kBouncer check. Note that kBouncer and ROPecker fail to detect gadgets longer than 20 and 6 instructions, respectively.

## 4.3 Heuristic Breakers

Since our exploit should fly under kBouncer's defensive radar, we have to ensure that we do not use sequences of more than seven short gadgets at any time (i.e., kBouncer-gadgets with 20 instructions or less). We avoid doing so by using a long gadget that performs minimal work (i.e., only sets a single register) as part of the exploit. Generally, to avoid the detection an attacker needs to intersperse the ROP chain with long gadgets. We call such gadgets *heuristic breakers* (HBs). The best properties for heuristic breaker gadgets are:

- **Use a small number of registers.** Such gadgets that preserve the values of registers allow us to chain multiple gadgets to carefully set the CPU and memory state to perform an operation like a call.

- **Used registers are loaded from memory or assigned constant values.** Long gadgets can have various side effects like loading and writing to memory, etc.. When the registers used in such operations are set within the gadget, it is easier to prepare the gadgets so that the gadget does not cause a fatal fault.

- **Registers are loaded from memory.** Gadgets including the epilogue of functions frequently restore the values of various registers from the stack, allowing us to set multiple registers from the controlled stack.

- **Intended gadgets.** Long sequences of unintended gadgets tend to translate to unusual sequences of instructions. As a result, they are not as useful as intended gadgets. Our exploit, uses only a single small unintended gadget of two instructions, while another one only uses an unintended call instruction so it is call preceded.

Finally, we have to be flexible when a HBs cannot be included at a desired position in the gadget-chain. Consider for example a chain of seven gadgets. Ideally, we would insert a HB after the first five gadgets to break the sequence in two smaller ones, of five and two gadgets respectively. Since this is not always possible, due to the exploit's semantics, we may need to insert a HB sooner, for example *after* the first three gadgets.

## 4.4 Putting It All Together

Figure 4 provides a high-level graphical representation of our PoC exploit. We obtain control by exploiting the button object's VFT pointer, which grants us control of an indirect jump instruction. This instruction is actually part of a gadget (Appendix A, listing 1), as far as kBouncer is concerned, however it is not a gadget for ROPecker because it contains a conditional branch. The end goal is to invoke `VirtualProtect()` to mark the buffer we control and contains shellcode as executable. We can then transfer control to it, effectively bypassing DEP and performing a code-injection attack.

Our first task is to point the stack pointer (i.e., `ESP`) to the buffer we control, so we can perform ROP, a process commonly referred to as stack pivoting. After receiving control, `eax` points to our buffer, so we use an unintended gadget that exchanges the values of `eax` and `esp`, and terminates with a `ret`, to achieve this (Appendix A, listing 2).

Next, we want to prepare for calling `VirtualProtect()`. Before doing so, we need to interpose a HB gadget, so that the kBouncer check, triggered by entering the API function, will not detect our exploit. At this point, we know that the LBR contains two gadget addresses, the one for the stack pivoting gadget and the one before that, which is part of the program's legitimate control flow. We know that these two gadgets are not enough to cause detection, but there may be other entries in the LBR preceding these that could trigger kBouncer. Using a HB at this point ensures that the gadget-chain length in the LBR is reset. Moreover, using a HB at this point makes the payload generic, allowing us to use it with other vulnerabilities, as it will always break the gadget chain in the LBR, as long as $T_G$ is less than its length. We use a HB gadget of 33 instructions that sets the `ESI` and `EDI` registers, which we use later on, and most importantly does not depend on any register being set up on entry (Appendix A, listing 3).

We perform the call to `VirtualProtect()` using a gadget that includes an indirect call (Appendix A, listing 4). This gadget only requires the `ESI` register to be prepared, which we set with the previous gadget. Also, it does not push any arguments to the stack, so the arguments to the call can be prepared in our buffer in advance. This is also the point where kBouncer kicks in and checks the LBR for an attack. By consulting Fig. 4, we notice that kBouncer cannot detect the attack at this point. When `VirtualProtect()` returns, control is transferred where it is expected to, that is, the instruction following the call. The next gadget executing is essentially the code following the indirect call (Appendix A, listing 5). The `ret` at the end of it transfers control to our shellcode, which is now executable. To ensure that we do not trigger any alarms in the future, we make sure that the first instruction in our shellcode is preceded by a fake, unused, call instruction.

Having managed to inject code into the process, we can now execute code without the risk of triggering kBouncer. Notice that this exploit will keep working even if $T_G$ is raised to 31 and $T_C$ reduced to 6.

# 5 Countermeasures

In this section, we discuss countermeasures, as well as fundamental boundaries in the use of gadget-chain length for preventing code-reuse attacks. While we mainly focus on kBouncer and ROPecker, we are confident that our analysis of their weaknesses and the proposed countermeasures will be of use to future works that plan to explore comparable methodologies.

## 5.1 Tweaking the $T_G$ and $T_C$ Parameters

An obvious improvement to both these techniques involves increasing $T_G$, i.e., the parameter that determines whether a sequence of instructions ending with an indirect branch is a gadget or not. Looking back at Fig. 4, it is clear that, in the case of kBouncer, increasing $T_G$ to 33 instructions would neutralize our exploit. However, increasing the length of gadgets is not straightforward, as it has various side effects.

Increasing $T_G$ will unavoidably lead to longer gadget chains that belong to legitimate, innocuous code. As we consider longer code sequence as potential gadgets, inevitably more application execution paths will be identified as gadgets, leading to observing longer gadget chains at run time. Consequently, to avoid false positives, $T_C$ also needs to be increased to avoid misclassifying legitimate control flows as attacks. Unfortunately, raising $T_C$ presents opportunities to attackers for using more gadgets. Both kBouncer and ROPecker assume that attackers cannot use longer gadgets due to the side effects that such gadgets have, something that both this paper and previous work [16] disproves. Defenders also face an asymmetry, as usual, because attackers need only find a handful of long gadgets to masquerade their payload. To maximize the effect of the parameters, what needs to be optimized is the fraction $\frac{T_G}{T_C}$. While this is probably an oversimplification, it provides a useful rule of thumb.

#### 5.1.1 Per-Application Parameters

Acceptable settings for $T_G$ and $T_C$ vary depending on the application being examined [8]. The nature of the application itself, the compiler it was built with, and the shared libraries it uses, influence the generated binary code and what parameter values can be used to avoid FPs and concurrently detect attacks consistently.

We can exploit this observation to use different values based on the application. This would ensure that the strictest rules are applied every time. However, doing so is also not without difficulty because both the defense and the attack depend greatly on the application in question. For example, after analyzing an application, we can determine that a very strict set of $T_C$ and $T_G$ can be used without FPs. However, the application may contain gadgets much larger than $T_G$ that can be directly chained together, so no gadget chains are identified. This scenario is obviously ideal for the attacker. Further research is required to establish a metric that quantifies the effect of selecting a particular set of parameters. Using per-application parameters can be also challenging in the presence of dynamically loaded (DL) libraries (i.e., libraries loaded at times other than program start up), as new, potentially unknown code is introduced in the application.

#### 5.1.2 Per-Call Parameters

A novel idea to further customize the parameters is to use different gadget-chain thresholds ($T_C$) based on the part of the code executing. kBouncer that triggers check on certain API calls, would greatly benefit from this approach. Certain APIs may be normally called through limited executions paths that exhibit very particular characteristics. For instance, a Windows native API call (win32) is frequently called by higher-level frameworks. This approach has the benefit of both avoiding FPs, hence providing better stability, and improving security guarantees.

#### 5.1.3 Cumulative Chain-Length Calculation

ROPecker also accumulates the lengths of smaller gadget-chain segments and uses a different parameter $T_{CC}$ to detect attacks. While this heuristic is not effective with our exploit, it would be interesting to explore whether incorporating it in kBouncer, which checks for attacks less frequently and in a more controlled manner, would further raise the bar for attackers.

#### 5.1.4 Obstacles

Recursive functions can cause significant problems with techniques based on counting gadget chains. Due to their nature, they can generate a large number of consecutive returns when they reach their end condition (e.g., when their computation has finished). If the returns within the recursive function lead to gadgets, then it is extremely hard to find any value of $T_C$ that would not cause FPs, unless the recursion is very shallow (relative to the value of $T_C$). kBouncer seems to avoid such conditions because it only checks the LBR when an API call is made. In a sense, it performs checks at the boundary between application and kernel, and the intuition is that the checks are made “far” away from the algorithms in the core of applications. However, it is not an uncommon scenario that a recursive algorithm requires to allocate memory or write into a file. In such cases, lowering the maximum gadget length ($T_G$) is the only option for avoiding FPs. On the other hand, ROPecker performs checks far more frequently and whenever execution is transferred to new pages, so we expect that it is even more fragile in the presence of recursive algorithms.

### 5.2 Combining with CFI

CFI [1] enforces control-flow integrity and can prevent the exploit we describe in Sec. 4. In particular, recent CFI approaches like CCFIR [39] and binCFI [40] prevent the use of unintended gadgets, such as the two-instruction gadget we use for stack pivoting (Appendix A, listing 2). CCFIR, in particular, also disallows indirect calls to certain API calls like `VirtualProtect()`, so it prevents two gadgets used by our exploit. These CFI approaches also incur low performance overhead, making them a good candidate for

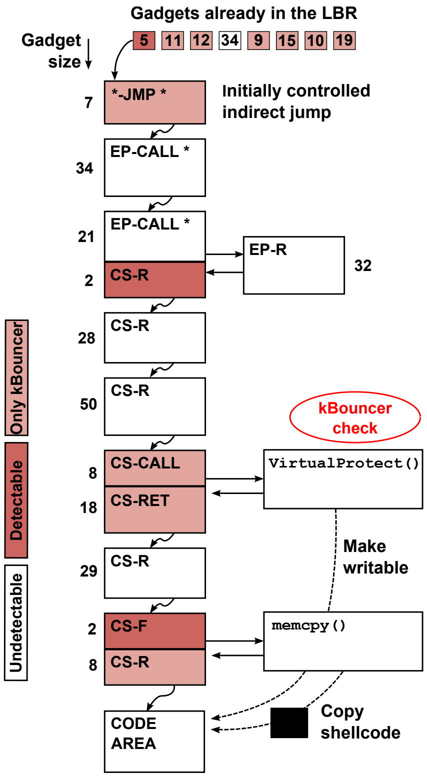


Figure 5: PoC exploit that bypasses kBouncer, ROPecker, and CCFIR. All the gadgets used are intended call-site (CS) or entry-point (EP) gadgets.

coupling with kBouncer. Unfortunately, recent work [16] has shown that they are still vulnerable to attack. *So is a combination of CFI and kBouncer still vulnerable?*

To answer the above question, we begin from the exploit used to bypass CFI in previous work [16, Sec. IV] and replace the smaller gadgets with longer HB gadgets that are also allowable by CFI. Similarly to prior work, we assume that CCFIR is in place, as it is stricter than binCFI. Because CFI does not allow transfers to new code, the goal of this payload is to mark existing code as writable and overwrite it with our shellcode. Before proceeding to describe the exploit, we summarize the additional restrictions imposed by CCFIR below.

Under CCFIR, return instructions can no longer transfer control to unintended gadgets, so only CS gadgets that were originally emitted by the compiler can be used when constructing a payload. Indirect call and jump instructions are also restricted and can only transfer control to function entry points, defining a new type of entry-point (EP) gadget. Indirect calls to sensitive API calls are prohibited, so any such calls need to be made using direct call instructions, contained within otherwise allowable gadgets. Finally, CCFIR introduces a new level of randomization through the use of springboard sections that proxy indirect control transfers. The location of these sections is randomized at load time and all indirect branches can only proceed through them.

To prepare the new payload, we need to replicate the steps described in Sec. 4, as well as a couple of additional steps required for bypassing CCFIR. Because of the randomized springboard sections, we need to reveal the sections that hold call and return stubs to the gadgets we plan to use. Fortunately, this can be achieved by exploiting the string object to leak code and meta-data from the DLLs of interest [16, Sec. IV.C]. `VirtualProtect()` and `memcpy()` are now called through gadgets that contain a direct call to these functions, so we do not need to explicitly locate them in the target process.

Figure 5 depicts a high-level overview of our second PoC exploit that overcomes the restrictions imposed by both kBouncer and CCFIR. We notice that because of CCFIR, we cannot use the same gadget to perform stack pivoting and, furthermore, we can only jump to an EP gadget. We resort to using a series of five gadgets to achieve the same goal (Appendix B, listings 6-10). Specifically, we first use three EP gadgets to corrupt the stack, so we can control a return instruction. This is similar to [16], however we use different, longer gadgets that are not detected by kBouncer. The fourth gadget loads `EBP` with our data and completes the switch to chaining through returns. We then use the fifth gadget, consisting of 28 instructions, to perform stack pivoting by copying `EBP` to `ESP` through the `leave` instruction.

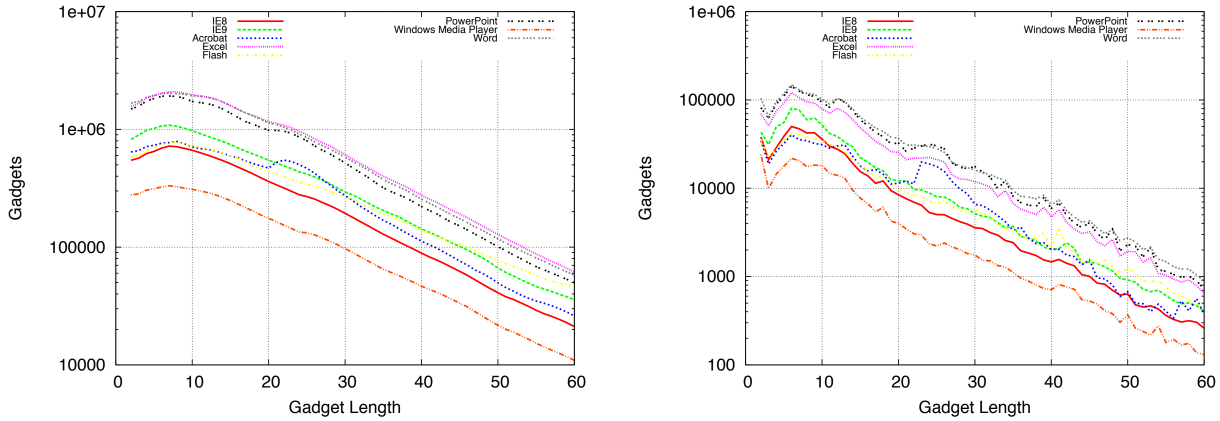
For calling `VirtualProtect()` and `memcpy()`, we reuse the same gadgets used against CFI (Appendix B, listings 12-13 and 15-16 respectively). The last gadget's `ret` transfers control to our shellcode that has been copied to the code section of the binary and has been preceded by a call to also foil future kBouncer checks. However, we replace the gadgets used to prepare the function-calling gadgets with the ones shown in listings 11 and 14 respectively, in Appendix B.

The exploit depicted in Fig. 5 demonstrates that even if we combine kBouncer and a loose CFI defense, it is still possible to devise attacks that can go undetected. Moreover, it shows that even if $T_C$ and $T_G$ are significantly tweaked, gadgets much larger than 21 instructions are available to partition an exploit to smaller, possibly undetectable, chains. In the particular exploit, setting $T_G$ to 33 and $T_C$ to 5 would still not have any effect.

# 6 Evaluation

## 6.1 Gadget Availability

The existence of long gadgets determines the potential to find HB gadgets that can be used to break long gadget chains. It is also an indicator on whether, we can poten-



(a) All gadgets.

(b) Call-preceded gadgets available to link using a `ret`.

Figure 6: Distribution of gadgets available to attackers under kBouncer based on their length, as recorded for multiple popular applications. We notice that there are numerous gadgets, even for gadgets of 60 instructions.

| **Application** | **Workload** |
|---|---|
| Windows Media Player | Music playback for approximately 30 secs |
| Internet Explorer 9 | Surf to google.com |
| Adobe Flash Player | Watch a YouTube video |
| Microsoft Word | Browse a Word document |
| Microsoft PowerPoint | Browse a PowerPoint presentation |
| Adobe Reader XI | Browse a PDF file |

Table 1: Applications used in the evaluation.

tially use many different HB gadgets with varying functionality. We analyze the applications listed in Tab. 1, along with all their DLLs, to determine how many gadgets of different sizes they contain. In addition, we analyze Internet Explorer 8, which we used in our PoC. We follow the same methodology we used for collecting gadgets for the PoC exploit (Sec. 4.2). Specifically, we developed a gadget extraction tool in Python, using the popular *distorm* disassembler [15].

The process begins by disassembling from each byte in the code segment of each target binary, recursively following conditional branches, direct calls and jumps to locate instruction paths that end with a return or an indirect call or jump. That is, potential gadgets. As we disassemble, we count the number of instructions on each of the traversed paths, while we also keep track of the nodes we visit to avoid counting the same instructions more than once, due to loops. If we find more than one path starting from a particular byte and ending in an indirect branch, we keep the shortest path, and consider its length to be the length of the gadget at that byte. This is in accordance to how kBouncer identifies gadgets.

Figure 6 draws our results. We notice that even for relatively large gadget sizes, there are tens of thousands of gadgets. While we cannot make any assumptions on how usable they are, these results are an indication that there is a significant pool of gadgets to choose from. We attribute their large number to the fact that unintended gadgets are basically allowed by kBouncer.

## 6.2 Per-Application Parameters

In this section, we evaluate the feasibility of our per-application parameter scheme described in Sec. 5.1.1. To determine if, indeed, different applications can benefit from using tighter parameters, we run the six applications listed in Tab. 1 performing simple tasks, such as browsing. These applications were also used to perform a similar evaluation in kBouncer. We follow the same methodology to measure gadget-chain length.

We use a run-time monitoring tool based on Intel's Pin [22] to emulate the operation of LBR. We monitor every indirect branch instruction, including returns, jumps, and calls, and log the running thread ID, the address of the branch, and its target. To locate kBouncer gadgets, we borrowed the scripts used by kBouncer to disassemble the application images and their DLLs and, at the same time, locate the sensitive API calls where checks are injected by kBouncer [25, Appendix]. We combine the statically and dynamically collected information to match gadgets with control transfers observed at run time and calculate the length of gadget chains that would be checked by kBouncer.

Figure 7 shows the size of gadget chains (for $T_G = 20$), as they would be stored in LBR when entering a sensitive API call and for different applications. We observe that among the tested applications, only Adobe Reader

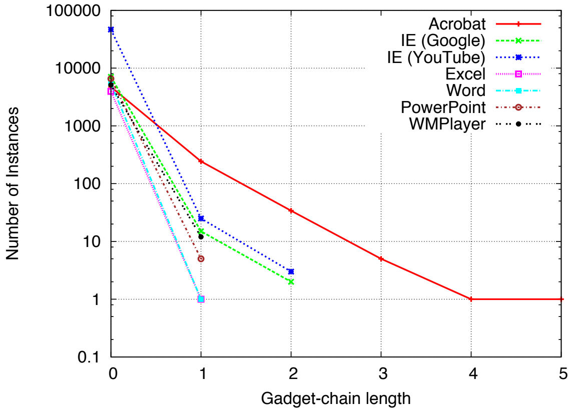


Figure 7: Length of chains for $T_G = 20$ for different applications, when entering a protected API function call. We run seven workloads using six applications, replicating the experiment in kBouncer [25, Sec. 3.2].

exhibits relatively long chains of five gadgets. This is in accordance with previously reported results [25]. All other tested applications include chains of up to two gadgets. In principle, this implies that we could use very strict values of $T_C$ for all these applications. For example, using $T_C = 3$ with Internet Explorer would detect our long five-gadget sequence preceding the call to `VirtualProtect()` (Fig. 4).

This experiment shows that application-specific parameters can make it very hard for attackers to evade detection, at least with the methodology followed by our PoC exploit. However, we remain cautious, as run-time analysis can have limited coverage, even when divergent workloads are used, so using such strict values for $T_C$ could cause FPs. In fact, recent works [12, 29] report FPs with applications other than the ones originally tested in kBouncer [25], and when more restrictive parameters are employed. Nonetheless, establishing viable values for these parameters through dynamic and static analysis calls for additional research. The above serve as an indicator that per-application parameter fitting is necessary.

## 7 Related Work

Code reuse is the dominant form of exploitation since the wide adoption of stack canaries [9] and data execution prevention [2], which provide protection against stack smashing and code injections respectively. Return-to-libc attacks [37] is one of the simplest types of code reuse, involving the redirection of control to a `libc` function after setting up its arguments in the stack. Usually, this involves invoking functions like `system()` or `exec()` to launch another program (e.g., to spawn a shell). Short gadgets were also used in register springs [10] to load a register with the address of the attacker-controlled buffer located in the randomized stack or heap.

Return-Oriented Programming [31] generalizes the task of leveraging existing code to compromise a program. Short snippets of code, called gadgets, are chained together to introduce a new, not initially intended, control flow. ROP is particularly effective on instruction sets like CISC, where there are no instruction alignment requirements and the instruction number is high, because any sequence of executable bytes in memory can potentially become a gadget. Nevertheless, RISC architectures are also vulnerable to ROP [5].

Diversification approaches like Address-Space Layout Randomization (ASLR) [26] can be effective against ROP attacks and are already present in most OSs. ASLR randomizes the layout of a program when it is executed by loading the binary and its dynamic libraries on different base addresses each time. ASLR can be brute-forced [32], but the difficulty of doing so increases as more entropy becomes available, like in 64-bit systems. Recent attacks bypassing ASLR [30] rely on memory disclosure bugs that leak enough data from the targeted process to infer where the binary and/or its libraries are loaded at run time.

Finer-grained randomization approaches [17, 24, 38] have been proposed to further diversify programs and limit the effectiveness memory leaks. In-place randomization [24] relies on randomizing the sequence of instructions and replacing instructions with others of equivalent effect to further diversify the image of a running process. ILR [17] attempts to break the linearity of the address space, and binary stirring [38] randomizes a binary in the basic block level. However, recent research [34] has demonstrated that bugs that allow an attacker to read almost arbitrary memory locations can be used to bypass the above solutions as well.

CFI [1] enforces control-flow integrity preventing the malicious control flows that are part of ROP attacks, and it is not affected by memory leaks. CFI requires an accurate control-flow graph of the target program, which usually implies access to source code, but recent works [39, 40] have made steps towards addressing this limitation by applying a loose version of CFI on binaries. However, it has been recently shown [16] that these loose-CFI approaches are still vulnerable to attack in the presence of memory leaks. This work builds on the latter, borrowing the notion of call-site and entry-point gadgets, which are also the only kind of accessible gadgets under kBouncer and friends, and uses the same IE vulnerability as a starting point. However, the attack described in [16] is not effective against kBouncer. In this work we build an attack that is again effective. We also show that using the LBR and a set of heuristics is not sufficient to prevent ROP attacks and reveal the inherent limitations of solutions based on gadget and chain length for detection. Finally, we propose various extensions that could alleviate the situation.

Compile-time solutions have been also proposed to alter the produced binary, so it is impervious to code reuse attacks. For example, producing a kernel that does not include any return instructions [21] cannot be exploited using ROP. However, variations of ROP that use indirect jumps instead of returns [3, 6] can be used to circumvent the above. G-Free [23] attempts to restrict control flow by using function cookies saved in a shadow stack at run time and inserts NOPs to destroy unintended gadgets. Because it is only loosely enforcing control flow, it is potentially vulnerable to the same attacks as CFI [16]. Additionally, compile-time approaches require that a binary and all of its libraries are recompiled.

Concurrently with our work, other efforts have also dealt with evaluating kBouncer and related approaches. Schuster et al. [29] take a slightly different approach and focus on finding gadgets that could be used to flush the LBR before performing any API call. The presence of such gadgets in the application nullifies any LBR-based defense, however, it leads to the same value being repeated in the LBR, which could potentially be used to detect such attacks. Moreover, the addition of CFI could restore the effectiveness of kBouncer.

On the other hand, Davi et al. [12] take an approach closer to ours. First, they show that there are enough small gadgets under loose CFI to perform any computation. Then, they introduce a long gadget of 23 instructions that does not perform any useful functionality and has limited side effects. They use this as a NOP gadget for breaking long gadget chains. Registers are not preserved, so additional gadgets need to be introduced to save any registers that need to be preserved. Larger NOP gadgets are not investigated, so unlike our approach their approach is more prone to detection when choosing stricter thresholds. Interestingly enough, both approaches test kBouncer, albeit with a different set of applications, and report false positives with the current, as well as with stricter thresholds.

## 8 Conclusion

In this paper we explored the feasibility of bypassing state-of-the-art ROP defenses based on monitoring processes (by means of Intel's new Last Branch Record) to detect control flows that resemble the execution of ROP chains [8, 25]. Essentially, these defenses check whether a sensitive API call was reached via a sequence of indirect branches to short, gadget-like, instruction sequences. Evading such detection is perceived as a hard task. First, all exploitation should be carried out using call-preceded gadgets, since otherwise the ROP chain will be easily detectable. Second, exploitation should find and then carefully insert long gadgets, in the middle of a series of shorter gadgets, in order to fly under the defense's ROP radar. The long gadgets should be long enough to make the ROP chain look like a legitimate control flow of the running process. Finding such long gadgets and gluing them in the actual ROP chain is not trivial, since it is possible that these long series of instructions interfere with the state of the exploit (e.g., modify a valuable register). Nevertheless, in this paper, we successfully constructed two real exploits, which utilizes the long gadgets to evade detection. With this work we stress that the selection of critical parameters, such as the length of a series of instructions that should be considered a gadget, as well as the gadget-chain length is not trivial. Until we solve these problems, the defenses are prone to false negatives and false positives. Finally, we discuss various countermeasures and provide evidence, through an experimental evaluation, that defining parameters on a per-application basis, can alleviate these concerns.

## Acknowledgment

We want to express our thanks to the anonymous reviewers for their valuable comments. In particular, we want to thank our shepherd, Kevin Fu, who helped us give this paper its final form. This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, or the Air Force. This work was also supported in part by the ERC StG project Rosetta, the FP7-PEOPLE-2010-IOF project XHUNTER, No. 273765, the Prevention of and Fight against Crime Programme of the European Commission – Directorate-General Home Affairs (project GCC), and EU FP7 SysSec, funded by the European Commission under Grant Agreement No. 257007.

## References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proc. of the 12th ACM CCS* (2005).

[2] Andersen, S., and Abella, V. Changes to functionality in Microsoft Windows XP Service Pack 2, part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

[3] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. of the 6th ACM ASIACCS* (2011).

[4] Bosman, E., and Bos, H. We got signal. a return to portable exploits. In *Proc. of the 35th IEEE S&P* (2014).

[5] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proc. of the 15th ACM CCS* (2008).

[6] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proc. of the 17th ACM CCS* (2010).

[7] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. DROP: Detecting return-oriented programming malicious code. In *Proc. of the 5th ICISS* (2009).

[8] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proc. of the 21st NDSS* (2014).

[9] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium* (1998).

[10] Dark Spyrit. Win32 buffer overflows (location, exploitation, and prevention). *Phrack magazine 9*, 55 (1999).

[11] DarkReading. Heap spraying: Attackers' latest weapon of choice. `http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428`, November 2009.

[12] Davi, L., Lehmann, D., Sadeghi, A.-R., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. of the 23rd USENIX Security Symposium* (August 2014).

[13] Davi, L., Sadeghi, A.-R., and Winandy, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proc. of the 6th ACM ASIACCS* (2011).

[14] DeMott, J. Bypassing emet 4.1. `http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf`, February 2014.

[15] distorm. Powerful disassembler library for x86/AMD64. `https://code.google.com/p/distorm/`.

[16] Göktaş, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Proc. of the 35th IEEE S&P* (May 2014).

[17] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. ILR: Where'd my gadgets go? In *Proc. of the 33rd IEEE S&P* (2012).

[18] HitmanPro. Real-time exploit mitigation and intrusion detection. `http://dl.surfright.nl/Alert-3/HitmanPro-Alert-3-Datasheet.pdf`, February 2014.

[19] Hunt, G., and Brubacher, D. Detours: Binary interception of Win32 functions. In *Proc. of the 3rd Conference on USENIX Windows NT Symposium* (1999).

[20] Intel. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide, part 2. `http://www.intel.com`.

[21] Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. Defeating return-oriented rootkits with return-less kernels. In *Proc. of the 5th EuroSys* (2010).

[22] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 26th PLDI* (2005).

[23] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th ACSAC* (2010).

[24] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. of the 33rd IEEE S&P* (2012).

[25] Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proc. of the 22nd USENIX Security Symposium* (2013).

[26] PaX Team. Address Space Layout Randomization, 2003. `http://pax.grsecurity.net/docs/aslr.txt`.

[27] Pelletier, A. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). VUPEN Vulnerability Research Team (VRT) Blog, July 2012. `http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php`.

[28] Portnoy, A. Bypassing all of the things. EXODUS INTELLIGENCE. `https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf`.

[29] Schuster, F., Tendyck, T., Pewny, J., Maass, A., Steegmanns, M., Contag, M., and Holz, T. Evaluating the effectiveness of current anti-ROP defenses. In *Proc. of the International Conference on RAID* (September 2014).

[30] Serna, F. J. CVE-2012-0769, the case of the perfect info leak. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

[31] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM CCS* (October 2007).

[32] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM CCS* (2004).

[33] Skowyra, R., Casteel, K., Okhravi, H., Zeldovich, N., and Streilein, W. Systematic analysis of defenses against return-oriented programming. In *Proc. of the 16th RAID* (2013).

[34] Snow, K. Z., Davi, L., Dmitrienko, A., Liebchen, C., Monrose, F., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of the 34th IEEE S&P* (May 2013).

[35] Sotirov, A. Heap feng shui in javascript. *Black Hat Europe* (2007).

[36] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proc. of the 2nd EuroSec* (2009).

[37] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. On the expressiveness of return-into-libc attacks. In *Proc. of the 14th RAID* (2011).

[38] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of the 2012 ACM CCS* (2012).

[39] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Proc. of the 34th IEEE S&P* (2013).

[40] Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *22nd USENIX Security Symposium* (2013).

# A PoC Exploit Gadgets

```
; library: mshtml.dll
; offset:  0x001BC907
; type:    intended, *-JMP *
mov   eax, [ecx+1Ch]
test  al,  al
js    loc1
```

```
loc1:
mov    ecx, dword ptr [ecx+24h]
mov    eax, dword ptr [ecx]
mov    edx, dword ptr [eax+24h]
jmp    edx                 ; edx points to 1st gadget
```

Listing 1: The exploitation of the vulnerability in Internet Explorer 8 gives us control of an indirect jump. The target address of the indirect jump is loaded from the sprayed buffer, by dereferencing values in the overwritten button object. After this code sequence, `ecx` contains a pointer to the overwritten button object, `eax` contains a pointer to our sprayed buffer, and `edx` contains the address of the first gadget.

```
; library: shell32.dll
; offset:  0x00146FB2
; type:    unintended, *-RET
xchg  esp, eax
retn
```

Listing 2: This is the first executed gadget after the control of the indirect jmp instruction. The gadget performs the stack pivoting operation. Essentially, the values in the `eax` and the `esp` registers are swapped. On entry, `eax` points to the sprayed buffer, which contains the rest of the ROP chain.

```
; library: shell32.dll
; offset:  0x0007AACD
; sort:    intended instructions, unintended CS-RET
mov     esi, 738AD720h
mov     edi, 73BCC3C0h
movsd
...
mov     esi, 738AD710h
mov     edi, 73BCC3D4h
movsd
...
mov     esi, 738AD730h
mov     edi, 73BCC3E8h
movsd
...
mov     esi, 738AD700h
mov     edi, 73BCC3FCh
movsd
...
mov     esi, 7387A2CCh
mov     edi, 73BCC410h
movsd
...
pop     edi
pop     esi
retn
```

Listing 3: This is a heuristic-breaker gadget, i.e., an undetectable long gadget. First, it is used to reset the chain of detectable gadgets in the ROP chain. Second, it will prepare the `esi` register, which is required by the next gadget that will call `VirtualProtect()`. Upon entry, it does not require any registers to be already set up, but it alters two registers: `esi` and `edi`, loading them with values from our buffer.

```
; library: shell32.dll
; offset:  0x0039C0E5
; type:    intended, CS-CALL *
lea     ecx, [esi+28h]
mov     edi, eax
mov     eax, [ecx]
call    dword ptr [eax+44h]
```

Listing 4: An indirect function call that we use to call `VirtualProtect()` and change the memory permissions of the region occupied by the injected shellcode, which also resides in the sprayed buffer. The gadget does not push values, so the arguments for the called function can be prepared in advance in the ROP chain, and it also saves `EAX` in `EDI` before calling.

```
; library: shell32.dll
; offset:  0x0039C0EF
; type:    intended, CS-RET
mov     eax, edi
pop     edi
pop     esi
pop     ebp
retn    0Ch
```

Listing 5: The instructions following the indirect call in listing 4 also constitute a gadget. This gadget restores `EAX` from `EDI`, thus restoring to the value it had before entering the previous gadget, and returns using the next value in our ROP chain transferring control to our shellcode.

## B CFI-resistant PoC Exploit Gadgets

```
; library: ieframe.dll
; offset:  0x00216C0E
; type:    EP
 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 sub     esp, 2C8h
 5 mov     eax, ___security_cookie
 6 xor     eax, ebp
 7 mov     [ebp-4], eax
 8 mov     eax, [ebp+0Ch]
 9 push    ebx
10 push    esi
11 push    edi
12 mov     edi, [ebp+8]
13 mov     [ebp-290h], eax
14 xor     eax, eax
15 push    3
16 mov     [ebp-280h], eax
17 mov     [ebp-284h], eax
18 mov     [ebp-288h], eax
19 mov     [ebp-2A0h], eax
20 mov     [ebp-2A4h], eax
21 pop     eax
22 mov     esi, ecx
23 mov     [ebp-2B8h], ax
24 mov     eax, [esi+24h]
25 lea     edx, [ebp-2C8h]
26 push    edx
27 mov     [ebp-2B0h], eax
28 mov     eax, [esi+1Ch]
29 mov     ecx, [eax]
30 lea     edx, [ebp-2B8h]
31 push    edx
32 push    eax
33 mov     [ebp-294h], edi
34 call    dword ptr [ecx+1Ch]
```

Listing 6: By pushing a pointer to the sprayed buffer as an argument (see Line 32), this gadget prepares the gadget (see Listing 7) that will call the stack smasher (see Listing 8).

```
; library: mshtml.dll
; offset:  0x004A959F
; type:    EP
 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 push    dword ptr [ebp+30h]
 5 mov     eax, [ebp+8]
 6 push    dword ptr [ebp+2Ch]
 7 mov     ecx, [eax+4]
 8 push    dword ptr [ebp+28h]
 9 mov     ecx, [ecx]
10 push    dword ptr [ebp+24h]
11 push    dword ptr [ebp+20h]
12 push    dword ptr [ebp+1Ch]
13 push    dword ptr [ebp+18h]
14 push    dword ptr [ebp+14h]
15 push    dword ptr [eax+0Ch]
16 push    dword ptr [eax+8]
17 push    dword ptr [ebp+10h]
18 push    dword ptr [ebp+0Ch]
19 push    eax
20 push    dword ptr [ecx]
21 call    dword ptr [ecx+10h]
```

Listing 7: This gadget will push the address of the call site gadget (see Line 15) we want to be executed later in the chain (see Listing 10). Once we get to this desired call site gadget, the switch from Entry Point to Call Site gadgets is complete.

```
; library: ieframe.dll
; offset:  0x000A98B5
; type:    EP
mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+8]
mov     ecx, [eax+140h]
push    ebx
mov     ebx, [ebp+14h]
push    esi
mov     esi, [ebp+0Ch]
mov     [esi], ecx
lea     ecx, [eax+144h]
mov     edx, [ecx]
push    edi
mov     edi, [ebp+10h]
mov     [edi], edx
lea     edx, [eax+148h]
mov     edi, [edx]
mov     [ebx], edi
xor     edi, edi
mov     [eax+140h], edi
mov     [ecx], edi
mov     [edx], edi
mov     eax, [esi]
neg     eax
sbb     eax, eax
pop     edi
and     eax, 7FFFBFFBh
pop     esi
add     eax, 80004005h
pop     ebx
pop     ebp
retn    10h
```

Listing 8: This is a long gadget that moves data and does not harm the status of our ROP chain. Also, it will break the calling assumptions of the caller gadget (see Listing 7).

```
; library: mshtml.dll
; offset:  0x004A95D6
; type:    EP
pop     ebp
retn    2Ch
```

Listing 9: The instructions following the indirect call in listing 7 also constitute a gadget. The return instruction in this gadget will use the address of the call site gadget that was pushed before (see Listing 7). Also, in this gadget ebp is prepared with a pointer to our sprayed buffer. The value in this register will be moved to the esp register in the stack pivoting gadget (see Listing 10).

```
; library: mshtml.dll
; offset:  0x00305202
; type:    CS
 1 mov     ecx, [ebp+30h]
 2 mov     edx, [ebp+40h]
 3 mov     [ecx], eax
 4 mov     eax, [ebp+34h]
 5 mov     ecx, [ebp+3Ch]
 6 shr     esi, 6
 7 and     esi, 1
 8 and     dword ptr [ebp+8], 0
 9 mov     [eax], esi
10 mov     eax, [ebp-0Ch]
11 mov     [ecx], eax
12 mov     eax, [ebp-4]
13 mov     ecx, [eax]
14 mov     ecx, [ecx+88h]
15 mov     [edx], ecx
16 mov     ecx, [eax]
17 mov     ecx, [ecx+30h]
18 mov     edx, [ebp+44h]
19 mov     [edx], ecx
20 mov     ecx, [ebp+38h]
21 mov     [ecx], eax
22 jmp     next_ins
23 mov     eax, [ebp+8]
24 pop     edi
25 pop     esi
26 pop     ebx
27 leave
28 retn    40h
```

Listing 10: This is a stack pivoting gadget. This gadget will load esp with a pointer to our sprayed buffer at Line 27.

```
; library: mshtml.dll
; offset:  0x0021FDF4
; type:    CS
mov     eax, [ebp+0Ch]
mov     [ebx], eax
push    7
pop     ecx
lea     esi, [eax+228h]
rep movsd
mov     ecx, [eax+244h]
mov     [ebx+20h], ecx
mov     ecx, [eax+260h]
mov     [ebx+24h], ecx
mov     ecx, [eax+264h]
mov     [ebx+28h], ecx
mov     ecx, [eax+268h]
mov     [ebx+2Ch], ecx
mov     ecx, [eax+26Ch]
mov     [ebx+30h], ecx
mov     ecx, [eax+270h]
mov     [ebx+34h], ecx
mov     ecx, [eax+274h]
mov     [ebx+38h], ecx
mov     ecx, [eax+278h]
mov     [ebx+3Ch], ecx
mov     ecx, [eax+27Ch]
mov     [ebx+40h], ecx
mov     ecx, [eax+280h]
mov     [ebx+44h], ecx
mov     ecx, [eax+284h]
mov     [ebx+48h], ecx
mov     ecx, [eax+288h]
mov     [ebx+4Ch], ecx
mov     ecx, [eax+28Ch]
mov     [ebx+50h], ecx
mov     ecx, [eax+290h]
mov     [ebx+54h], ecx
mov     ecx, [eax+2CCh]
mov     [ebx+58h], ecx
mov     ecx, [eax+2D0h]
mov     [ebx+5Ch], ecx
mov     ecx, [eax+2D4h]
mov     [ebx+60h], ecx
mov     ecx, [eax+2D8h]
mov     [ebx+64h], ecx
mov     eax, [eax+2DCh]
pop     edi
mov     [ebx+68h], eax
pop     esi
mov     eax, ebx
pop     ebx
pop     ebp
retn    8
```

Listing 11: This is a long Heuristic Breaker gadget. It will also prepare the ebp and ebx registers which are required by the VirtualProtect calling function (see Listing 12).

```
; library: ieframe.dll
; offset:  0x0006FBAE
; type:    CS
and     dword ptr [ebp-0Ch], 0
lea     eax, [ebp-0Ch]
push    eax                 ; old protection
push    40h                 ; new protection
push    ebx                 ; size
mov     ebx, [ebp-8]
push    ebx                 ; address
call    ds:VirtualProtect
```

Listing 12: Gadget that makes program code writeable, whereto inject a shellcode later. The part after the call instruction is considered as a separate gadget as it is the target of an indirect branch (i.e., return instruction of the VirtualProtect function).

```
; library: ieframe.dll
; offset:  0x0006FBC3
; type:    CS
test    eax, eax
jz      loc_766E9531 ; if eax==0: handle error
mov     eax, [ebp+8]
and     dword ptr [edi+4], 0
mov     [edi+8], eax
mov     [edi+10h], esi
mov     [edi+0Ch], ebx
mov     eax, [ebp-0Ch]
mov     [edi+14h], eax
mov     eax, dword_768E2CCC
mov     [edi], eax
mov     dword_768E2CCC, edi
xor     eax, eax
pop     edi
pop     ebx
pop     esi
leave                 ; == mov esp, ebp and pop ebp
retn    14h
```

Listing 13: Gadget that makes program code writeable, whereto inject a shellcode later. The part after the call instruction is considered as a separate gadget as it is the target of an indirect branch (i.e., return instruction of the VirtualProtect function)

```
; library: mshtml.dll
; offset:  0x000DA72B
; type:    CS
mov     eax, [ebp+10h]
mov     [ebx+108h], esi
mov     esi, [ebp+8]
lea     edi, [ebx+110h]
movsd
...
mov     esi, [ebp+0Ch]
push    0Dh
pop     ecx
lea     edi, [ebx+120h]
rep movsd
mov     [ebx+154h], eax
mov     eax, [ebp+1Ch] !!
mov     ecx, [eax]
push    0Dh
mov     [ebx+0C0h], ecx
pop     ecx
lea     esi, [eax+18h]
lea     edi, [ebx+0C4h]
rep movsd
mov     eax, [eax+4Ch]
pop     edi
pop     esi
mov     [ebx+0F8h], eax
pop     ebx
pop     ebp
retn    18h
```

Listing 14: Another Heuristic Breaker gadget and at the same time it will prepare eax for the memcpy calling gadget.

```
; library: ieframe.dll
; offset:  0x001ADCC2
; type:    CS
push    eax                 ; destination
call    memcpy
```

Listing 15: Gadget for calling of memcpy for copying the shellcode to existing program code.

```
; library: ieframe.dll
; offset:  0x001ADCC8
; type:    CS
add     esp, 0Ch
xor     eax, eax
jmp     short loc_7672DCE7
pop     ebx
pop     edi
pop     esi
pop     ebp
retn    8
```

Listing 16: The call site part of the memcpy calling gadget.