Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets

Andrea Biondo
University of Padua, Italy
andrea.biondo.1@studenti.unipd.it

Mauro Conti
University of Padua, Italy
conti@math.unipd.it

Daniele Lain
University of Padua, Italy
dlain@math.unipd.it

***Abstract*—Attackers use memory corruption vulnerabilities to compromise systems by hijacking control flow towards attacker-controlled code. Over time, researchers proposed several countermeasures, such as Address Space Layout Randomization, Write XOR Execute and Control Flow Integrity (CFI). CFI is one of the most promising solutions, enforcing control flow to adhere to statically determined valid execution paths. To trade with the execution and storage overhead, practical CFI implementations enforce coarser version of CFI. One of the most widely deployed implementations of CFI is the one proposed by Microsoft, named Control Flow Guard (CFG). CFG is currently in place on all Windows operating systems, from Windows 8.1 to the most recent update of Windows 10 (at the time of writing), accounting for more than 500 million machines.**

In this paper, we show a significant design vulnerability in Windows CFG and propose a specific attack to exploit it: the Back to The Epilogue (BATE) attack. We show that with BATE an attacker can completely evade from CFG and transfer control to any location, thus obtaining arbitrary code execution. BATE leverages the tradeoff of CFG between precision, performance, and backwards compatibility; in particular, the latter one motivates 16-byte address granularity in some circumstances. This vulnerability, inherent to the CFG design, allows us to call portions of code (gadgets) that should not be allowed, and that we can chain together to escape CFG. These gadgets are very common: we ran a thorough evaluation of Windows system libraries, and found many high value targets – exploitable gadgets in code loaded by almost all the applications on 32-bit systems and by web browsers on 64-bit. We also demonstrate the real-world feasibility of our attack by using it to build a remote code execution exploit against the Microsoft Edge web browser running on 64-bit Windows 10. Finally, we discuss possible countermeasures to BATE.

I. Introduction

Memory corruption vulnerabilities are a prime tool for attackers [40]. The typical course of an attack to exploit such vulnerabilities involves hijacking the program's control flow to execute arbitrary code in the application's context. There exist many different countermeasures to mitigate the impact of such attacks. One of the most widely implemented techniques is Address Space Layout Randomization (ASLR) [29], which aims at preventing the attacker from gaining crucial information about the program's memory structure. However, researchers showed that ASLR is vulnerable to application-specific information leaks [35] along with OS-based [6] and hardware-based [15], [18] side-channels. Another popular mitigation is Write XOR Execute (W⊕X), also called Data Execution Prevention (DEP) [26] in Windows. W⊕X aims at thwarting code injection and modification by enforcing that every page of memory may be either writable or executable, but not both. However, W⊕X can be bypassed by various code reuse techniques [5], [34], [36]. Indeed, albeit these mitigations can make exploiting memory corruption significantly harder, they do not stop modern multi-stage attacks.

A promising mitigation mechanism is Control Flow Integrity (CFI) [4], which stems from the idea of effectively restricting the program's control flow to only valid paths that the programmer expected. This stops the attacker from diverting execution to unintended code paths. Ideal CFI requires precise and time-consuming program analysis, based on pointer analysis that is undecidable in the general case [30], and can have significant performance overheads. For this reason, in the the last decade, researchers mostly put effort in building CFI implementations that balance security, performance and practicality [7]. Practical CFI implementations often adopt an *approximation* of ideal CFI: they only enforce a superset of the actual valid execution paths. Such approximations need careful balancing, as the tradeoff is usually between security guarantees and cost (both in terms of memory and computational power). Indeed, while approximation makes a relaxed version of CFI affordable for real-world applications and raises the bar for exploitation, it still leaves enough leeway for attacks that can leverage this imprecision to subvert control flow [13], [17], [34], thus obtaining arbitrary code execution. Also, implementations can themselves be vulnerable to memory corruption [10]. Practical CFI implementations that are widely deployed in the real world are, for example, Indirect Function-Call Checks [43] in LLVM and Clang, Virtual Table Verification [43] in GCC, and Control Flow Guard (CFG) [23] in Microsoft Visual C++. Microsoft CFG is one of the most popular implementations: it was introduced in Windows 8.1 for applications compiled with Microsoft Visual C++. At the time of writing, it is deployed on at least 500 million computers running Windows 8.1 through the latest Windows 10 versions, and is considered to be an important stage of defense against memory corruption attacks.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23318
www.ndss-symposium.org

In this paper, we show a serious vulnerability caused by a significant interaction between a design assumption of Windows CFG and the Windows libraries. CFG assumes functions that can be target of indirect calls (e.g., through a virtual table) will be aligned to 16 bytes. That is, the code of such functions should always start at a 16-byte aligned boundary in memory. However, the compiler does not always enforce the correct alignment. When this happens, the approximated nature of CFG allows us to reach portions of code (gadgets) that are not intended to be marked as valid targets. Among these possible gadgets, we define a particular set of *pop-ret (PR) gadgets*, and a novel class of gadgets that we call *spiller (S) gadgets*. We exploit PR gadgets to completely evade CFI and employ traditional non CFI-aware exploit techniques. PR gadgets require a limited form of stack control, which can be easily obtained on 32-bit systems via controlled arguments. S gadgets can be used to regain this ease of control on 64-bit, as exploitation on this platform is more challenging than on 32-bit. We propose a novel attack that uses these gadgets to evade CFG: the Back To The Epilogue (BATE) attack[1].

Previous attacks on CFG relied on calling (now disabled) sensitive APIs [9], on stack control [32], [33] or on application-specific issues [16], [39], [50]. BATE only requires control of an argument to a corrupted indirect call. Moreover, we only require PR (and possibly S) gadgets to be present in a single library loaded in the target process. We show that this is often the case: we ran a complete assessment of gadget availability in system libraries. On 32-bit we found that PR gadgets are widespread: in particular, we note their presence in C/C++ runtime libraries, which are often loaded as they an essential part of the environment compiled C/C++ code runs in. On 64-bit systems, the number of available gadgets is more limited; however, they are in libraries that are appealing to attackers, such as the legacy JavaScript engine and a media codec. We found PR gadgets in another very popular software: 64-bit Microsoft Office 2016. We found that S gadgets are similarly widespread.

BATE has real-world capabilities. To demonstrate this, we built a remote code execution exploit against Microsoft Edge on 64-bit Windows 10 with ASLR, DEP and CFG mitigations. At a high level, the exploit employs two vulnerabilities in the Chakra JavaScript engine [22]: an information leak [2] and a type confusion [3]. The information leak is used to bypass ASLR, which is a necessary stepping stone for further exploitation. The type confusion allows us to corrupt C++ virtual tables and launch the BATE attack. BATE bypasses CFG and hijacks the program's control flow to arbitrary locations. While this is sufficient to prove that our attack works, we go through with the exploit to show that code execution is possible. To bypass DEP, we start with stack pivoting [11]: we redirect the stack pointer to a fake stack that contains a first-stage ROP chain. This ROP payload makes a second-stage shellcode executable and transfers control to it, achieving our goal.

[1]We responsibly disclosed the BATE attack to Microsoft, who acknowledged it and is working on a countermeasure. As part of a coordinated disclosure roadmap, we obtained permission to submit this work.

Contributions. The contributions of this paper are the following:

- We identify a design weakness in Control Flow Guard, Microsoft's CFI implementation: the assumption that targets are always 16-byte aligned. This allows invalid targets to become valid, when a valid target function is not 16-byte aligned, causing a portion of code of the previous function to become a valid target as well.
- We show a practical way of exploiting this vulnerability: the Back To The Epilogue (BATE) attack. BATE leverages two sets of gadgets: *pop-ret (PR) gadgets* that require a limited form of stack control, and *spiller (S) gadgets* that help us use PR gadgets on 64-bit systems. PR gadgets, in particular, can be found in function epilogues, can be unintended valid targets thanks to the aforementioned vulnerability, and can be used to completely bypass CFG.
- We extensively and thoroughly analyze system libraries of recent builds of Windows 10, and of a very popular software: Microsoft Office 2016. We perform pattern matching and symbolic execution, and find numerous occurrences of PR and S gadgets (for example in C/C++ runtime libraries, JavaScript engines, media codecs, and libraries used by the Microsoft Office suite). This proves that BATE is a real threat: its gadgets are contained in libraries that are commonly loaded by third-party software (such as web browsers), and therefore likely available to attackers.
- We further prove the real-world feasibility of this attack by using our PR and S gadgets in a remote exploit against Microsoft Edge, on 64-bit Windows 10 with all security features enabled, that allows us to execute arbitrary code on the victim's machine.
- We discuss possible countermeasures to protect CFG from BATE and propose a short-term solution that stops BATE without redesigning CFG.

Organization. This paper is organized as follows: we first report related work in Section II. In Section III we give general background on Control Flow Integrity and describe the internals and weaknesses of Control Flow Guard. We discuss our considered threat model in Section IV. We detail our attack in Section V, and assess its impact in Section VI. We demonstrate a practical real-world exploit in Section VII. We discuss our attack and outline possible countermeasures in Section VIII. Section IX concludes the paper.

II. Related Work

Here, we present related work. First, we cover general CFI techniques and vulnerabilities, along with some implementation examples. Then, we discuss specific works on Control Flow Guard (CFG), Microsoft's CFI implementation, and its weaknesses.

A. Control Flow Integrity

Control Flow Integrity (CFI) is a security policy that enforces adherence between a program's statically determined

control flow graph and its runtime execution path. It aims at preventing attackers from diverting execution to paths that were not intended by the programmer. Forward-edge CFI protects forward branches, such as indirect calls through function pointers or virtual tables, which could be hijacked by an attacker. Backward-edge CFI protects from corruption of return addresses, for example via a shadow stack [12]. One of the main characterizing factors of a CFI approach is its granularity. Fine-grained CFI aims at restricting control flow to the exact program's control flow graph, keeping a separate set of allowed destinations for each indirect call site. However, the pointer analysis required for ideal CFI is undecidable [30], so a certain level of approximation is inevitable. Coarse-grained CFI is more relaxed and enforces a global set of valid targets.

The seminal work on CFI [4] proposes an approach based on labeling indirect call destinations. Indirect calls are instrumented to check whether the destination label is the expected one, to ensure that an edge for that transfer exists in the control flow graph. While they suggest that more label classes can make exploitation harder, their implementation only uses a single class and is therefore coarse-grained. Backward-edge protection is achieved via a shadow stack, whose integrity is guaranteed by CFI.

Researchers have proposed multiple different approaches, with varying granularites and performance characteristics [7]. For example, CCFIR [49] works directly on binaries and achieves coarse-grained, forward- and backward-edge CFI by forcing all returns and indirect calls to go through aligned stubs in a springboard section. CCFIR is more precise than the original CFI, since it separates backward edges in two equivalence classes. Two open-source compilers, GCC and LLVM, also offer forward-edge CFI implementations [43]. GCC's Virtual-Table Verification (VTV) protects from virtual table hijacking attacks [31] by checking that the virtual table belongs to the class hierarchy for the invocation object. LLVM's Indirect Function-Call Checks (IFCC) protects all indirect calls by redirecting them through jump tables. IFCC can support many levels of precision. The authors focus on two: all functions allowed for any call, or grouped by their number of arguments. Particularly interesting to our paper is MIP [28], another coarse-grained implementation, as the data structure is close to Microsoft CFG. It defines a mapping between memory addresses and bit positions in a bitmap and divides the code into chunks, only the beginning of which are valid targets. Bitmap bits corresponding to the beginning of chunks are set, while others are cleared. Indirect calls and returns are instrumented to extract the bit for the target address from the bitmap and check whether it is the beginning of a chunk.

Based on the observation that points-to analysis always has some level of approximation, Evans et al. show a technique to exploit the imprecision of scalable analyses [14]. Coarse-grained CFI implementations suffer from attacks that leverage their big equivalence classes. While they limit the scope of code reuse attacks, since most code locations are not valid targets, they are not immune. Indeed, it is still possible to build realistic and Turing-complete gadget sets under CFI policies [13], [17]. A powerful attack is COOP [34]: it derives gadgets from C++ virtual methods, which are valid CFI targets, and chains them by exploiting virtual table hijacking. However, this attack leads to a restricted gadget set, which makes it more difficult to write payloads. BATE achieves full instruction pointer control, allowing an attacker to follow up with simpler exploit techniques. Additionally, Conti et al. [10] demonstrate weaknesses in multiple CFI implementations when the attacker is in control of the stack.

B. Control Flow Guard.

Microsoft Control Flow Guard (CFG) [23] is a forward-edge, coarse-grained CFI implementation based on a bitmap, similar to the approach proposed by MIP [28]. CFG internals are not officially documented, but third parties reverse-engineered them [27], [41], [46]. We describe how CFG works in more detail in Section III-B. Instead, we now report past exploits and bypasses to CFG, grouped by the weakness they exploited.

Some bypasses rely on finding code that is not protected by CFG, as CFG allows all branches to modules that are compiled without CFG support [27]. Thus, code from those modules can be reused. As more and more modules are compiled with CFG, this becomes less of a problem.

Another source of vulnerabilities is Just-In-Time (JIT) compiled code. By default, dynamic allocations of executable memory are not protected by CFG, allowing code reuse attacks such as JIT spraying [38], [47]. To avoid this problem, the implementation has to take care of properly altering the bitmap to mark valid targets in dynamic code. Moreover, JIT compilers have to replicate the CFG instrumentation in the generated code to protect outgoing branches. For example, Falcón [16] shows unprotected indirect calls from the Flash JIT compiler.

Due to its coarse granularity, CFG does not distinguish between call sites, and keeps a global set of allowed targets. An attacker can exploit this by calling valid target functions, that are however unintended for that particular code path. For example, certain Windows APIs change the execution context and can be abused to hijack the control flow [9]. However, the introduction of *sensitive APIs* [20] has sensibly reduced the number of allowed dangerous APIs, thus decreasing the attack surface for such bypasses.

CFG also relies on certain assumptions about memory protection. Checks are performed by calling CFG functions provided by the operating system via function pointers. Those pointers are filled in by the kernel and reside in a read-only memory area, so it should be impossible to corrupt them. However, some application-level bugs can be used to make memory writable, allowing an attacker to overwrite the pointers [50]. BATE is more general as it does not target a specific bug. Moreover, researchers found that this area can actually be writable in some modules [39]. Furthermore, CFG only protects forward-edge transfers. Since it does not protect the return address on the stack, an attacker can overwrite it to gain flow control [33].

Finally, Windows stores pointers to exception handling routines on the stack. Branches to those functions are not

protected by CFG, so an attacker can hijack them and then cause an exception to transfer control to arbitrary locations [32].

III. Background

Before presenting our BATE attack against CFG, we need to recall some basic concepts. Here, we first introduce the main concepts of Control Flow Integrity (Section III-A). We then explain how Control Flow Guard works (Section III-B).

A. Control Flow Integrity

Control Flow Integrity (CFI) [4] is a security policy that aims at preventing adversaries from redirecting control flow to arbitrary locations. Many CFI implementations have been proposed in the literature, with varying degrees of precision and performance [7]. CFI computes the application's control flow graph statically, either from source during compilation or directly from binaries. At runtime, control flow is monitored to ensure it sticks to the computed graph. This can be done by performing checks on instructions that trasfer control, to ensure they have not been corrupted by an attacker. Control transfers can be divided in forward (calls and jumps) and backwards (returns), based on the direction of their edge in the control flow graph. Depending on what kind of transfers are protected, CFI can be *forward-edge*, *backward-edge* or both. Forward-edge CFI protects jumps and calls, which can be *direct* or *indirect*. Direct branches embed their destination in the instruction itself. Assuming that executable memory is not writable, which can be ensured by W⊕X, those cannot be corrupted by an attacker. Indirect branches load their destination from a memory location, for example when calling through a function pointer. An attacker that can overwrite the code pointer in memory can redirect the branch. In Figure 1 we show an example of such attack, and how CFI can prevent it. In this example, Figure 1a shows code without CFI enforcement, leading to a successful control flow hijack. Figure 1b, on the other hand, shows CFI enforcement and detection of the attack.

In more detail, in Figure 1a a memory corruption vulnerability is used to overwrite `fptr2` and hijack the second call to `evil` instead of the intended target `func3`. Indirect calls are extremely common in object-oriented code, where they are used to implement virtual method calls through virtual tables. Many modern attacks rely on virtual table corruption, in order to easily divert control flow [31]. To avoid this, CFI checks indirect branches at runtime to ensure that the computed graph has an edge for the transfer. This can be done by statically determining the set of valid targets for a call, i.e., the *points-to set* of the function pointer, and then checking the destination against it at runtime. Figure 1b shows the same attack as before, this time with CFI. The only allowed target for `fptr1` is `func1`, while `fptr2` can point to `func2` or `func3`. Indeed, before the calls, there are CFI checks. A call to `evil` through `fptr2` violates the CFI policy, so the corruption is detected and can be handled, typically by killing the process.

Unfortunately, precise and sound points-to analysis is hard and, in the general case, undecidable [30]. As such,

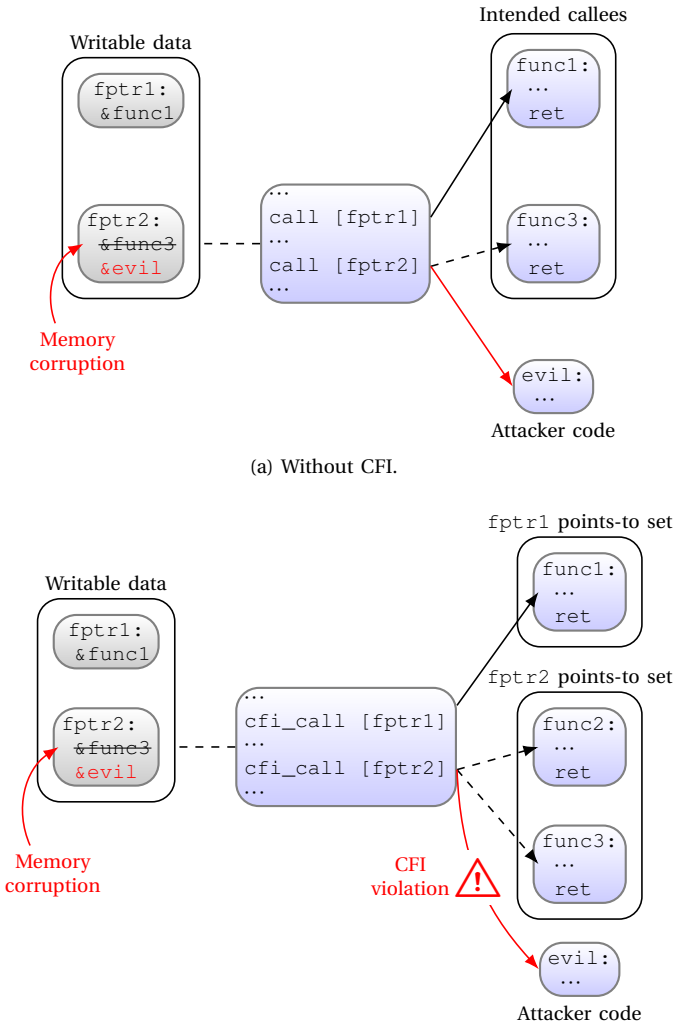


Fig. 1. Example of function pointer hijacking, with and without CFI techniques in place. If there is no CFI, control flow goes to the pointer specified by the attacker; if CFI is in place, attacker would be likely pointing to an invalid target, and CFI successfully prevents malicious redirection.

the set of allowed targets is an approximation of the actual one. To indicate the level of approximation, CFI policies are either *fine-grained* or *coarse-grained*. While the meaning of these terms is not standardized in the literature, we use fine-grained CFI to mean a policy where each call site has a distinct valid target set, which is as precise as possible. In other words, only paths that the programmer intended can be taken. In coarse-grained CFI, there is a single valid target set, consisting of all the targets of indirect branches in the entire program. This clearly extends the attack surface, as an adversary can call unintended functions. For example, if a coarse-grained CFI was used in Figure 1b, `func{1,2,3}` would all be valid targets for both `fptr1` and `fptr2`. There are also CFI schemes that employ an intermediate number of equivalence classes: for example, IFCC [43] can group valid targets by number of arguments.

Backward-edge CFI protects return instructions. When a call is performed, the address of the next instruction in the caller is stored on the stack, to be later used as a

return address. Attacks such as stack overflows allow an adversary to overwrite the return address, gaining control of the execution flow when the callee returns. Statically determining the set of valid return locations is not very precise, as a function can be called from many different places. For this reason, backward-edge CFI implementations often make use of a *shadow stack* [12], which resides in a protected memory area and stores a copy of the real return address. On returns, the return address fetched from the real stack can be compared to the one from the shadow stack and the program can detect whether it was tampered with.

B. Control Flow Guard

Researchers and companies provided several practical CFI implementations. One of the prominent ones, because of widespread diffusion in all Windows operating systems from 8.1 onwards, is Control Flow Guard (CFG). CFG is a coarse-grained forward-edge CFI implementation that leverages an instrumentation involving the compiler, the kernel and the `ntdll.dll` library [46]. It protects from hijacked forward branches such as function calls and `longjmp` buffers, but does not offer backward-edge CFI, which guards return addresses. CFG relies on a process-wide bitmap to perform fast integrity checks, which is similar to the approach used by MIP [28].

We describe in more detail in the following how CFG works: we start from code analysis and compiler instrumentation at compile time (Section III-B1), then explain the core *bitmap* employed technique (Section III-B2), and finally detail how runtime integrity checks work (Section III-B3).

1) Compiler instrumentation: When building a module (executable or library), the compiler analyzes the source code and generates a *valid target table* for indirect branches, that is, the set of all entry points that can be valid indirect targets. By default, this analysis includes exports to support dynamic symbol resolution. However, a feature named *export suppression* allows the programmer to make specific exports invalid targets. The valid target table is embedded into the binary's read-only data section. The compiler also sets up two global function pointers in the same section, for *check* and *dispatch* functions. Those pointers will be later filled in by the kernel when loading the module and pointed to implementations for CFG checks. Both functions take an indirect branch target and check whether it is valid. If this is the case, the check function simply returns, while the dispatch function jumps to the target. Otherwise, both terminate the process with a security violation. To support pre-8.1 Windows, the compiler provides dummy implementations with which the pointers are initialized, to be later overridden by the loader. The compiler uses those functions to implement two modes:

1) *check mode*, where a call to the check function is injected before indirect branches;
2) *dispatch mode*, where indirect calls are replaced by a call to the dispatch function.

Both modes use the same checking algorithm and are equivalent for our purposes. For 32-bit modules, the compiler can insert further checks to ensure that the stack pointer does not change after the indirect call [45], [46]. This mitigates stack desynchronization attacks based on mismatching calling conventions [17]. Dispatch mode, which is common on 64-bit [20], is only distinguished by the implementation performing the target branch instead of the caller.

2) Module loading: When the kernel loads a CFG-aware module, it fetches the valid target table and encodes this information into the *CFG bitmap*, a continuous block of read-only reserved virtual memory (32MB on 32-bit, 2TB on 64-bit) in the process' addressing space. Each pair of bits in the bitmap bijectively maps to a 16-byte aligned address range of 16 bytes in size, so that every address in user space maps to one and only one bit pair. Thus, each range can have one of four states associated with it:

- **`00`** - no address in this range is a valid target;
- **`01`** - this range contains an export-suppressed target;
- **`10`** - the only valid target is 16-byte aligned (that is, the first address in the range);
- **`11`** - all addresses in this range are valid.

The only virtual bitmap pages actually backed by physical memory are those that are not completely zeroed. Moreover, since even when countermeasures such as Address Space Layout Randomization are in place dynamic libraries are only relocated at their first load, bitmap regions for libraries used by multiple processes can share their physical backing. As such, the committed memory footprint is acceptable. If a module is not CFG-aware, all the bit pairs belonging to its address space will be set to `11`. This allows intermodule calls from a module that employs CFG to one that does not, which is essential to preserve backwards compatibility. The loader also points the check and dispatch function pointers to implementations within `ntdll`.

3) Runtime: After loading a module, its bitmap region is not necessarily static. It can be altered in two ways:

1) by allocating executable memory, whose bitmap bits will all be set;
2) by changing specific bits through system calls.

A typical case that requires bitmap modification is when code is generated via a just-in-time compiler. Any change is local to the process, so modifying a shared bitmap page will result in a private copy being mapped.

To clarify the CFG mechanism, we show in Figure 2 how call checks happen at runtime (in *check* mode). In this example, the `fptr` function pointer resides in a writable data section, so it could be vulnerable to corruption. The compiler has protected an indirect call to it via check mode by prepending a call to the CFG check function. First, the system fetches the indirect target from `fptr` and stores it into the `rcx` register, which is where the check function expects it to be. Then, the check function is called indirectly. This is safe as long as the check and dispatch pointer are read-only. The check call will jump into `ntdll`. Here, the position of the bit pair (highlighted in the figure) for the target address into the bitmap is calculated via fast bitwise operations and the bits are fetched. If the pair resides in an

unmapped bitmap page, a memory violation exception will occur, which is handled by a top-level handler in `ntdll`. The handler contains a special case that checks whether the violation happened within the CFG checking code. If it did, then the check is resumed as if the page was completely set to zero, which will lead to a failure since `00` means that there are no valid targets. The target address is checked against the bit pair to determine whether it is valid. In the example the pair is `10`, which means the check will pass only if the target is 16-byte aligned. If the check fails the process is terminated, otherwise the check function returns to the caller, which finally issues the original indirect call. In dispatch mode, the target address would be passed in the `rax` register and the call would be issued directly by `ntdll`.

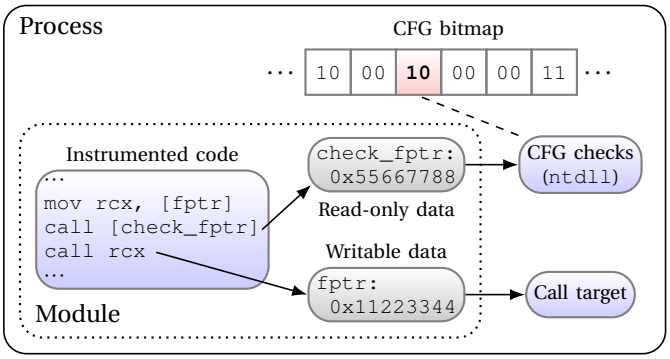


Fig. 2. Example of CFG *check mode* of a function call: first, the *check function* in `ntdll` is called and the bit pair from the bitmap is checked; if the value from the bitmap allows the call target, the check function returns and the function is called. Otherwise, a security violation is issued.

IV. Threat Model and Assumptions

Overall, our threat model is even stricter than what is considered in the CFI seminal work [4]. We assume the application's control flow integrity protection is provided solely by CFG: therefore, other CFI implementations and other integrity protections, such as VTint [48], are not in place. Regarding offensive capabilities, our attacker is less powerful than usual: we assume the attacker has knowledge of the memory layout and a limited form of memory corruption, but we do not require arbitrary write capability. Regarding defensive capabilities by the defender, differently from the threat model considered in [4], we consider memory layout randomization to be in place, as it is nowadays a common countermeasure.

Offensive Capabilities. The attacker has the following capabilities:

- **Memory layout knowledge.** The attacker can know the layout of the program's addressing space, for example by reading pointers from memory.
- **Indirect call corruption.** The attacker can leverage some memory corruption vulnerability in the program to hijack the destination of indirect calls.
- **Control near the stack top.** The attacker can control a word near the top of the stack. We show that this can be achieved by controlling an argument to a corrupted indirect call, both on 32-bit and 64-bit systems.
- **Adversarial computation.** The attacker can perform runtime calculations, for example by targeting a scripting language interpreter such as JavaScript.

While the seminal work assumed that the attacker had total control over memory contents, our attack only requires the ability to corrupt indirect calls. The stack control requirement does not imply further memory corruption needs, as we gain it via controlled arguments.

Defensive Capabilities. The following defenses are in place:

- **W⊕X.** By default, every memory mapping is either writable or executable, but not both. This stops an attacker from modifying code (because it is not writable) or injecting code (because it is not executable).
- **Randomization.** The memory layout of the program is randomized, for example via Address Space Layout Randomization (ASLR).

V. BATE: our Attack to CFG

In this section we describe our Back To The Epilogue (BATE) attack to CFG. We start with an overview of BATE (Section V-A), followed by a discussion of the specific weaknesses we exploit (Section V-B). Then, we define *pop-ret* gadgets (Section V-C), which are a central part of our technique, and how we exploit them to gain flow control on 32-bit systems (Section V-D). We then introduce *spiller* gadgets (Section V-E), which work as helpers for pop-ret gadgets on 64-bit. Finally, we show to combine pop-ret and spiller gadgets to mount a 64-bit attack (Section V-F).

A. Overview

At the high level, BATE works by exploiting a design assumption about the alignment of valid CFG targets. Whenever target functions are not correctly 16-byte aligned, we are able to jump to code that surrounds the entry point of these functions. In particular, we jump to code sequences we call *pop-ret* (PR) gadgets, that are contained within the epilogue of a function preceding a valid unaligned target. Those gadgets modify the stack pointer and allow us to transfer control to any location, provided that we control a value reasonably close to the top of the stack.

The described attack can be easily done on 32-bit code by controlling an argument to an hijacked indirect call. On 64-bit, controlling a value near the stack top is more difficult than on 32-bit. To make PR gadgets work, we need control over a zone on the stack known as Register Parameter Area (RPA). For this reason we introduce *spiller* (S) gadgets, which spill attacker-controlled values to this area. We then combine S and PR gadgets to form an S-PR chain, which gives us flow control. Since many S gadgets take the spilled values from arguments, we regain the ease of exploitation via arguments we had on 32-bit.

B. Exploited Weaknesses

We exploit three weaknesses in CFG: presence of unaligned targets coupled with design assumptions about target alignment, lack of backward-edge protection, and that the bitmap is process-wide.

Unaligned Targets. CFG is able to precisely mark a valid target only if it is the only target in its address range and it is 16-byte aligned. In that case, the state will be `10`. However, if a target is not aligned, or there are multiple targets in the same range, then the state will have to be set to `11`, which allows branches to any address in the range. In other words, we can freely alter the lower 4 bits of a valid unaligned target and the result will still be a valid target. This enables us to reach code located near a unaligned function's entry point, which leads to interesting code sequences we call PR gadgets. Note that if multiple targets are in the same range at least one has to be unaligned, so we will not distinguish between the two and just refer to unaligned targets. This design assumption would not impact CFG's security if the compiler always aligned targets. However, in practice, we were able to find unaligned targets in code commonly used by applications.

Lack of backward-edge CFI. CFG does not check return addresses. Our technique eventually returns to an attacker-controlled value, thus successfully escaping from CFG. We achieve return address control by modifying the stack pointer before a return instruction. As previously noted, 32-bit CFG can include stack pointer checks. However, we take control of execution before these checks run, effectively neutering them.

Process-wide bitmap. Since CFG keeps a single bitmap for the whole process, a valid target is allowed for any indirect branch in any module. Thus, we can improve the feasibility of our attack by extending the search for gadgets to all loaded modules. This is particularly interesting when gadgets are in system libraries: all processes that load a library with our gadgets in it automatically become exposed to BATE. Since system libraries can potentially be loaded by a large number of applications, we get a more universal bypass.

C. PR gadgets

Due to the imprecision around unaligned targets, we can jump in the neighborhood of a valid unaligned function's entry point. We now need to look for sequences of instructions that perform interesting operations from an attacker's point of view. At the low level, a function is typically made of three parts:

1) **Prologue.** It spills (i.e., saves) callee-saved registers to the stack and sets up the stack frame for the function.
2) **Body.** It performs the actual work defined by the programmer at the higher levels.
3) **Epilogue.** It deallocates the stack frame and restores callee-saved registers.

Since a function's prologue is placed at its entry point, and the compiler lays out functions one after the other in the binary, the epilogue of a function is close to the entry point of the subsequent function. We show an example of this situation in Figure 3: `func2`, at `0x1007`, is a valid CFG target but it is not 16-byte aligned. Therefore, the entire aligned 16-byte range (`0x1000-0x100f`, represented by the shadowed cells in the picture) around `func2` is valid. This code range, between dashed lines in the figure, includes the epilogue of `func1`, or at least a part of it.

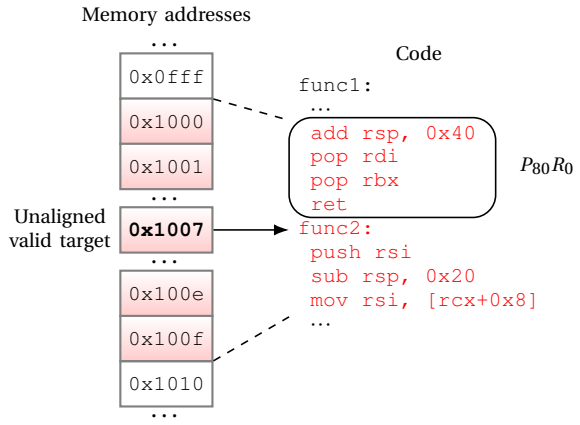


Fig. 3. An unaligned valid target at `0x1007` (`func2`) makes the whole `0x1000-0x100f` range valid. A $P_{80}R_0$ gadget is generated from the epilogue of `func1`.

A typical epilogue performs the following operations:

1) **Stack frame deallocation.** The stack frame was allocated in the prologue by subtracting its size from the stack pointer. Deallocation is done by either adding the stack frame size to the stack pointer, or by setting the stack pointer to the base pointer, which keeps track of the stack frame base. The latter can only be used if the base pointer is not being treated as a general-purpose registers, which is a common optimization.
2) **Register restoration.** The original values for callee-saved registers were pushed to the stack during the prologue. The epilogue pops them to restore their value for the caller.
3) **Return.** The `ret` instruction pops the return address from the stack and branches to it. There is also an alternative `ret` opcode that accepts a 16-bit immediate operand, which will be added to the stack pointer after popping the return address.

All operations done by the epilogue, excluding deallocation if the base pointer is used, increment the stack pointer by a fixed amount. A *pop-ret (PR) gadget* is a sequence of consecutive instructions that increment the stack pointer and return. Due to the predictability of compiler-generated epilogues, it is easy to define the exact structure of PR gadgets generated from epilogues. Each PR gadget is described by two parameters, p and r, denoted as P_pR_r, and satisfies the following properties:

- **PR.1** The gadget is a valid target for CFG.
- **PR.2** The gadget is composed by the following sequence of instructions:

1) An optional `add {e,r}sp, m` instruction. If not present, let $m = 0$.
2) An optional sequence of n `pop` instructions, excluding `pop {e,r}sp`, since it would change the stack pointer to a value that is not necessarily controlled. If not present, let $n = 0$.
3) Either a `ret` instruction, in which case let $r = 0$, or a `ret` r instruction.

- **PR.3** $p = m + wn \geq w$, where w is the native word size in bytes (4 on 32-bit, 8 on 64-bit).

In our example in Figure 3 the epilogue of `func1`, which is reachable because `func2` is an unaligned valid target, generates a $P_{80}R_0$ gadget ($w = 8$, $m = 64$, $n = 2$). PR gadgets are *relative* ROP stack pivots: they increment the stack pointer by p bytes and return. Optionally, they can increment the stack pointer by another r bytes before returning, but after popping the return address.

The main insight behind BATE is that PR.1 is often satisfiable. Since we exploit the imprecision around unaligned targets, PR gadgets must be in the 16-byte CFG range for the unaligned target, otherwise they will fail the CFG check. More precisely, since PR gadgets precede the unaligned target, our search window is restricted to the lower part of the range, which extends from the lowest address in the range (i.e., the unaligned target rounded down to a multiple of 16) to the unaligned target, excluded. Assuming an uniform distribution for the unaligned targets, this window will on average be 8 bytes. However, `pop` and `ret` instructions are small (1 or 2 bytes for `pop`, 1 or 3 bytes for `ret`), so useful PR gadgets can be very short and fit into this space. Moreover, a single epilogue can generate multiple PR gadgets. For example, the epilogue of `func1` in Figure 3 also contains the subsequences `pop/pop/ret` and `pop/ret`, which are respectively $P_{16}R_0$ and P_8R_0 gadgets.

Since every instruction in a PR gadget increments the stack pointer, p grows as we get farther from `ret`. Big pivots typically come from having an `add` instruction, which is before all `pop`s, and therefore quite far from the return. However, the distance between the gadget entry point and `ret` is limited, because the entry point has to lay within the lower part of the 16-byte CFG range for the unaligned valid target in order to pass CFG checks. Assuming the offsets of unaligned targets within their ranges are uniformly distributed, we can expect gadgets with big p values to be rarer than ones with small p. We show that this is indeed the case in Section VI.

D. Exploiting PR gadgets

To further explain how to use PR gadgets, we refer to the sample stack we depict in Figure 4. The x86 stack grows backwards, towards lower memory addresses. We refer to the lowest address, which the stack pointer points to, as the *top* of the stack.

Figure 4a shows the stack layout immediately after a `call` instruction for 32-bit calling conventions [24]. Before the call, the caller pushes the arguments to the stack, bringing the stack pointer to sp_0. The return address is then pushed to the stack before branching to the call target, so that the new stack pointer is $sp_1 = sp_0 - w$. For our attack, we corrupt an indirect call and redirect it to a P_pR_r gadget. The gadget will increase the stack pointer by p bytes before returning. Let $sp_r = sp_1 + p$ be the stack pointer when the gadget reaches its `ret` instruction. By PR.3 we have that $sp_r \geq sp_0$, meaning that the return address will be fetched from the caller's stack frame. Return addresses are not checked by CFG, so by controlling this location one can make the program branch to an arbitrary destination. Once execution jumps to the attacker's target, the stack pointer will be $sp_r - w + r$. As an example, consider Figure 4b, where an indirect call has been hijacked to a $P_{2w}R_r$ gadget. The gadget will take its return address from the second argument to the callee, which is an attacker-controlled value within our threat model.

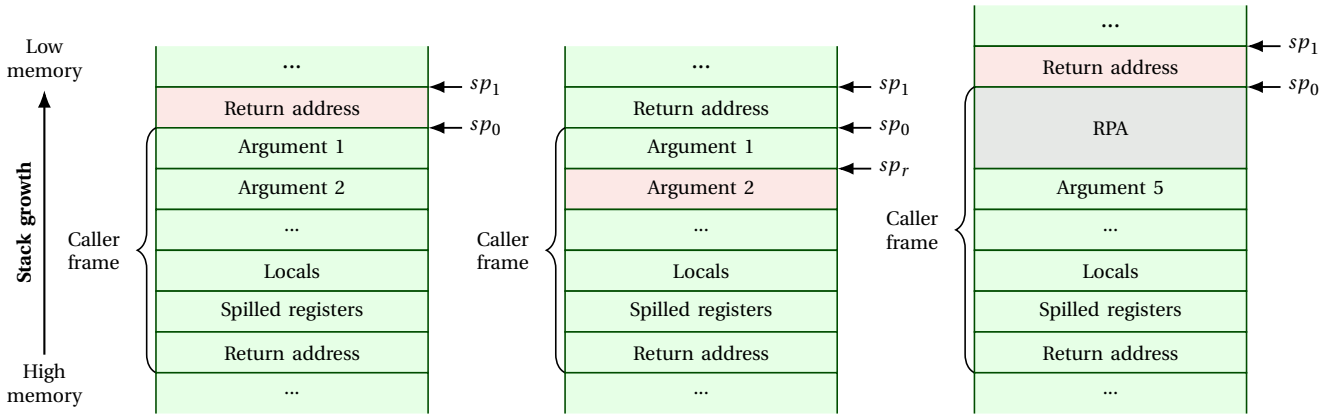
Clearly, a PR gadget with a big p value could set the return address further down the stack frame, for example in the local variables or in the registers spilled by the caller, both of which an attacker might be able to control. However, big pivots are rarer than small ones, so we focus on controlled arguments as they are closer to the stack top. We also note that, since PR gadgets pop registers, an attacker that controls more than just the word at sp_r can use them to control registers as a side effect. While we do not make use of this in our proof-of-concept, it can aid exploitation since follow-up techniques such as stack pivoting often require a controlled register.

Difficulties on 64-bit. While this technique works on 32-bit code, it is not as easy to apply on 64-bit. Figure 4c shows the stack layout after a `call` instruction for the Microsoft 64-bit calling convention [25]. The first four arguments are passed in registers and subsequent ones via the stack. This immediately reduces the impact of PR gadgets, because many functions do not take more than four arguments. Also, a Register Parameter Area (RPA) is inserted at the top of the caller's stack frame. The RPA is allocated by the caller for the callee to spill registers into, and is 4 registers (32 bytes) in size. Reaching below this zone would require PR gadgets with $p \geq 40$, which are rare. Therefore, we aim at using PR gadgets that pivot into the RPA, which requires control over its contents.

E. S gadgets

The RPA is typically used for argument registers, although other registers may be spilled into it. When arguments are spilled, they are in left-to-right order in memory. While the RPA can be used by the caller for temporary storage between calls, in most case it is left untouched. Driving sp_r into the RPA would access uninitialized stack data.

While stack data used before initialization can be controlled, for example as shown for the Linux kernel [21], it requires a complex setup and significant effort. Instead, we exploit a common compiler optimization to control the RPA: the replacement of tail calls with tail jumps. A function ending in a tail call would have that call as the last instruction in its body, followed by the epilogue and the return. This is often optimized by first executing the



(a) 32-bit stack layout after `call`.

(b) 32-bit stack layout after a $P_{2w}R_r$ gadget.

(c) 64-bit stack layout after `call`.

Fig. 4. On 32-bit, the stack pointer after a call (sp_1) is close to the arguments and the caller's locals, so the stack pointer when a PR gadget returns (sp_r) can be pivoted into attacker-controlled arguments. On 64-bit bit, the register parameter area is uninitialized and distances sp_1 from the caller's stack frame.

epilogue and then simply jumping to the callee. When control reaches the callee, the stack pointer points to the caller's return address and the callee will build its stack frame over what was the caller's stack frame. This reduces stack depth and avoids an extra return, since the callee will directly return to the caller's caller.

Most importantly, the caller likely spilled its arguments, or in some cases other general purpose registers, to the RPA. Since the stack frame has been deallocated, the stack at the tail jump looks again like Figure 4c, but now the RPA contains initialized data. Let us assume that the tail call is indirect, but CFG-protected, and that we can hijack it. This puts us in a position where we can chain a PR gadget with an initialized RPA near the stack top. To apply this technique in practice, we find functions with tail call optimization that are also valid CFG targets, which we name *spiller (S) gadgets*. An S gadget is described by a parameter n, denoted as S_n, and satisfies the following properties:

- **S.1** The gadget is a valid target for CFG.
- **S.2** The gadget spills n registers to the RPA.
- **S.3** The gadget ends with a controlled indirect tail jump after its epilogue.
- **S.4** The gadget has negligible or manageable side effects.

This is an approximate notation, that does not take into account which registers are spilled and at what offsets. We give a more precise description of S gadget semantics in Section VI. We note that S gadgets can be considered as a particular subclass of the EP-IJ gadgets defined in [17]. If the final jump is made through a virtual table, they are also similar to COOP [34] gadgets. We redirect the final indirect jump of an S gadget to a PR gadget to build an *S-PR chain*.

F. Exploiting S-PR chains

Control flow can be hijacked by redirecting an indirect call for which we control a spilled register to an S-PR chain with a PR gadget that will pivot sp_r to the spilled value. We focus on argument registers (`rcx`, `rdx`, `r8`, `r9`) because they are easier to control, but we stress that other registers may be spilled.

Figure 5 shows a realistic example in a C++ application. The first code chunk from the top makes an indirect call via dispatch mode, which takes the target address in the `rax` register. This particular example shows a C++ virtual call: `rcx` (first argument) is the `this` pointer, the virtual table pointer is at offset 0 in the object and the method pointer is at offset 0x50 in the virtual table. The attacker has corrupted the virtual table pointer so that `rax` is loaded with the address of an S_2 gadget. Also, the attacker controls `rdx`, which is the second argument, and sets it to the address of the final target.

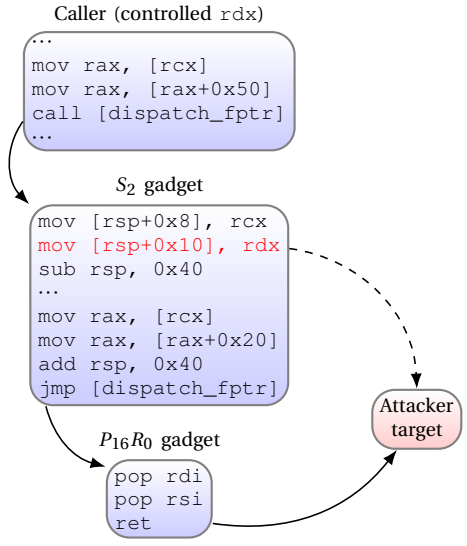


Fig. 5. An S-PR chain, that shows how S gadgets are used to setup the stack for PR gadgets.

In this example, the S gadget starts by spilling the first two arguments (`rcx` and `rdx`) to the RPA, which begins

at `rsp+8`. In particular, the highlighted instruction spills the attacker-controlled `rdx` to `rsp+16`. The gadget then builds a stack frame (64 bytes), performs some innocuous operations, and prepares `rax` for an indirect dispatch-mode tail call. This is again a virtual call, on the same corrupted object as before, at a 0x20 offset. Thus, the attacker can hijack it to a $P_{16}R_0$ gadget. Finally, due to the tail call optimization, the S gadget deallocates the stack frame and jumps (via CFG dispatch) to the PR gadget. The stack pointer is now the same as it was at the beginning of the S gadget, and the spilled `rdx` is again at `rsp+16`. The PR gadget increments the stack pointer by 16 bytes, bringing the spilled register to the top of the stack, where `ret` will use it as the return address, achieving flow control.

VI. Impact and Applicability

We systematically assessed the presence of PR and S gadgets both in system libraries (as they expose to BATE all applications that load them), and Microsoft Office 2016, a very popular software and thus high-value target. Results were obtained from clean installs of 32-bit and 64-bit Windows 10 Pro Insider Preview, build 16232.1000.

A. Analysis

To find PR gadgets we apply a simple pattern matching approach, as their structure is predictable. For each file, we extract the list of valid targets and filter it down to unaligned targets. Then, we disassemble backwards from unaligned targets, at every offset within the window of addresses that share all but the lowest 4 bits. We match the disassembled chunks against the PR gadget structure previously illustrated, to determine whether we found a PR gadget and what its semantics are. This works well because the layout of epilogues is fixed.

For S gadgets we use a less naive approach, because they are much more diverse. To simplify exploitation, we only identify S gadgets made of a single basic block, without any control flow transfer except the final indirect jump. We extract the valid targets from the file, then we disassemble forwards until we reach an instruction that changes control flow, such as a call, a jump or a return. If this instruction is an indirect jump to the dispatch pointer, or if it is a call to the check pointer followed by an indirect jump to a register, we mark the gadget as a candidate. We then symbolically execute candidate gadgets to determine whether the stack frame has been deallocated prior to the indirect branch, and what was spilled to the RPA.

We use Capstone [1] for disassembly and angr [37] for symbolic execution.

B. Results

Table I shows libraries that contain unaligned targets and PR gadgets. We recall that, in a P_pR_r gadget, p indicates how many bytes are added to the stack pointer before returning, and r how many are added after returning. As expected (see discussion in Section V-C), small p values are more frequent than big ones. We also note that, as a general trend, p grows at first by steps of 8 bytes (from `pop` instructions), then suddenly increases because of `add`s.

TABLE I. Unaligned targets and PR gadgets found in Windows 10 system libraries.

Library	Unaligned targets	Total PR gadgets	PR gadgets (deduplicated)
32-bit Windows, 32-bit WoW64 subsystem			
AppVEntSubsystems32.dll	1	—	—
clusapi.dll	1	—	—
d3dim.dll	322	1	P_4R_0
d3dim700.dll	323	11	$P_4R_{\{0,8\}}$, $P_8R_{\{0,8\}}$, $P_{12}R_8$, $P_{16}R_8$, $P_{80}R_8$
msvcr120_clr0400.dll	17	5	$P_4R_{\{0,4098\}}$, P_8R_0, $P_{12}R_0$, $P_{36}R_0$
msvcrt.dll	34	15	$P_4R_{\{0,4,8\}}$, P_8R_0, $P_{12}R_0$, $P_{16}R_4$, $P_{36}R_0$, $P_{40}R_0$, $P_{44}R_0$, $P_{52}R_0$
MSVP9DEC.dll	40	10	P_4R_0, P_8R_0, $P_{12}R_0$, $P_{16}R_0$, $P_{20}R_0$, $P_{112}R_0$
MSVPXENC.dll	53	11	P_4R_0, P_8R_0, $P_{12}R_0$, $P_{16}R_0$, $P_{20}R_0$, $P_{112}R_0$
ntdll.dll (32-bit only)	1	—	—
resutils.dll	1	—	—
ucrtbase.dll	6	4	P_4R_0, P_8R_0, $P_{12}R_0$, $P_{36}R_0$
user32.dll	3	—	—
wsp_fs.dll	1	—	—
wsp_health.dll	1	—	—
64-bit Windows			
jscript9.dll	9	4	P_8R_0, $P_{16}R_0$, $P_{24}R_0$, $P_{32}R_0$
msmpeg2vdec.dll	1	3	P_8R_0, $P_{16}R_0$, $P_{56}R_0$
MSVPXENC.dll	1	—	—
PayloadRestrictions.dll	5	—	—
rtmpltfm.dll	6	4	P_8R_0, $P_{16}R_0$

On 32-bit systems our bypass is widely applicable, because PR gadgets can be found in C/C++ runtime libraries (such as `msvcrt.dll`), which are loaded by most applications, along with being dependencies for a large number of other system DLLs. The same applies to 32-bit applications on 64-bit systems, which run through the WoW64 subsystem. On 64-bit the attack surface is smaller. However, we found two libraries that offer PR gadgets and are particularly appealing to attackers: `jscript9.dll` is the legacy JavaScript engine used by Internet Explorer, while `msmpeg2vdec.dll` is a system video codec that could be loaded by applications that handle media files. Analyzing the 64-bit Microsoft Office 2016 suite, we found 1410 unaligned targets in 139 executables and libraries, resulting in 123 non unique PR gadgets. Of those, 101 are $P_{40}R_0$, which are particularly interesting as they reach beyond the RPSA. We do not report details of these gadgets because of space issues.

We show an approximate overview of S gadgets on 64-bit Windows in Table II. Both the Internet Explorer (`jscript9.dll`) and Edge (`Chakra.dll`) JavaScript engines contain a fair number of S gadgets. The same holds for the HTML parsers used by the two browsers (`mshtml.dll` for Internet Explorer and `edgehtml.dll` for Edge). We also note the presence of S gadgets in real-time codecs used by Skype and graphics libraries. We highlight that this is only an approximate overview, as our notation is not completely precise: we define n as the total number of spilled 64-bit registers. We do not take into account 32-bit

registers, as they are generally not useful. Most importantly, the S_n notation does not describe exactly which registers are spilled, whether they are argument registers or not, and at what position in the RPA they are spilled. We report a precise description of the S gadgets we found and of the registers they spill in Appendix A.

TABLE II. S GADGETS FOUND IN WINDOWS 10 64-BIT SYSTEM LIBRARIES.

Library	Total S gadgets	S gadgets (deduplicated)
aadtb.dll	3	S_1
Chakra.dll	52	S_1, S_2, S_3
Chakradiag.dll	1	S_2
CoreUIComponents.dll	1	S_1
d2d1.dll	1	S_1
d3d10warp.dll	1	S_1
D3DCompiler_47.dll	64	S_1, S_2, S_3, S_4
dbghelp.dll	76	S_1, S_2, S_3, S_4
edgehtml.dll	76	S_1, S_2, S_3
FlashUtil_ActiveX.dll	2	S_1
jscript9.dll	34	S_1, S_2, S_3
jscript9diag.dll	5	S_2, S_3
mrt_map.dll	3	S_4
mshtml.dll	217	S_1, S_2, S_3, S_4
msvcp120_clr0400.dll	41	S_1, S_2, S_3, S_4
msvcr120_clr0400.dll	12	S_1
ortcengine.dll	28	S_1, S_2, S_3, S_4
pdm.dll	24	S_1, S_2, S_3, S_4
pidgenx.dll	2	S_3
rgb9rast.dll	4	S_1, S_2
rometadata.dll	3	S_1, S_2, S_3
rtmcodecs.dll	12	S_1, S_2
rtmmvrortc.dll	2	S_1
rtmpal.dll	83	S_1, S_2, S_3, S_4
rtmpltfm.dll	129	S_1, S_2, S_3, S_4
sppc.dll	6	S_1, S_2, S_3
sppcext.dll	1	S_2
SystemSettings.Handlers.dll	7	S_1
SystemSettingsThresholdAdminFlowUI.dll	12	S_1, S_2, S_4
Windows.Media.Protection.PlayReady.dll	20	S_1, S_2, S_3
Windows.UI.Input.Inking.Analysis.dll	58	S_1, S_2, S_3
WsmSvc.dll	5	S_1, S_2

In summary, our attack is feasible against most applications on 32-bit systems and against high-value targets (such as web browsers and very popular applications) on 64-bit.

An important question related to BATE is the reason why the compiler does not align some targets. Indeed, this interplay between misalignment and CFG's assumption creates the vulnerability that BATE exploits. To investigate this, we analyzed an already compiled library: the 64-bit `jscript9.dll`, as it is based on ChakraCore, which is open source. We observed that the compiler ignores some alignment directives for handwritten assembly routines, causing the misalignments in `jscript9.dll`. We speculate that it might correspond to a number of reasons, such as a compiler bug that manifests in borderline cases, or a bug in Microsoft's compilation pipeline. However, we could not replicate misalignment by compiling neither ChakraCore nor some custom test code. Moreover, our analysis is limited to a single library, and the source of unaligned targets in other libraries may be different.

VII. PROOF OF CONCEPT IMPLEMENTATION OF BATE

To demonstrate BATE, we build a proof-of-concept remote code execution exploit against the Microsoft Edge web browser running on 64-bit Windows 10 Anniversary Update. We exploit two known vulnerabilities in the Chakra JavaScript engine: an information leak [2] and a type confusion [3]. These vulnerabilities are already used in a public proof-of-concept exploit [42], on which we draw inspiration for our implementation. However, the public exploit overwrites a thread's stack to hijack control flow, thus needing arbitrary write on the stack. We remark that BATE allows us to obtain control and bypass CFG without the need of arbitrary write, making our technique more general.

The outline of our proof-of-concept exploit is as follows. We first discover the layout of the program's memory to locate our gadgets. We then corrupt a C++ virtual table to redirect execution into an S_2-$P_{16}R_0$ chain and gain flow control. This step highlights that our technique bypasses CFG. Finally, to present a working proof-of-concept exploit (even if we already bypassed CFG), we use stack pivoting and ROP to bypass DEP and execute arbitrary code. We now describe the exploit in more detail.

A. Primitives

We build two primitives from the vulnerabilities: the first one allows us to leak the absolute address of an arbitrary JavaScript object. The second one provides us with arbitrary memory read/write. We stress that, differently from the public exploit, we use the arbitrary write in a very limited way: we only corrupt heap objects, but not other memory areas such as the stack. Therefore, we could exploit less "powerful" vulnerabilities with our attack.

Address leak. The information leak vulnerability allows us to leak the addresses of the elements of an array. We leverage this into a primitive that leaks the address of an object, by constructing an array that contains the object and then using the information leak to get the address.

Arbitrary read/write. We use the type confusion vulnerability to confuse an array and a `DataView` object. By altering the array, we can change the data pointer the `DataView` works on and perform memory reads and writes from it.

B. Gadget selection

We use a $P_{16}R_0$ gadget from `msmpeg2vdec.dll` (Figure 6) and a S_2 gadget from `chakra.dll` (Figure 7). The latter is already loaded in memory, since it belongs to the JavaScript engine. It will spill its second argument (`rdx`) to `rsp+16` and call the function at offset 0x50 in the virtual table of the object pointed to by the first argument (`rcx`). To bring the PR gadget into memory, we embed an MPEG-2 video in the exploit page, which forces the `msmpeg2vdec.dll` codec to be loaded. When chained to the S gadget, it will return to the second argument of the S gadget.

C. ASLR bypass

Since we will have to hijack indirect calls to the gadgets, we need to know their absolute addresses, which are randomized due to ASLR. To derandomize them, we find the

```
1  ; @ msmpeg2vdec+0xb29c
2  pop rdi
3  pop rsi
4  ret
```

Fig. 6. The $P_{16}R_0$ gadget from `msmpeg2vdec.dll` used in our proof-of-concept exploit.

```
1  ; @ chakra+0x31f0000
2  chakra!ScriptEngine::EnumHeap:
3  mov r11, rsp
4  ; Spill arguments to RPA
5  mov [r11+0x10], rdx
6  mov [r11+0x8], rcx
7  ; Allocate stack frame
8  sub rsp, 0x28
9  ; Prepare call to rcx->__vfptr[10]
10 mov rax, [rcx]
11 mov r8, rdx
12 xor edx, edx
13 mov rax, [rax+0x50]
14 ; Deallocate stack frame
15 add rsp, 0x28
16 ; Perform indirect call via CFG
17 jmp cs:__guard_dispatch_icall_fptr
```

Fig. 7. The S_2 gadget from `chakra.dll` used in our proof-of-concept exploit.

base of the containing modules, which then can be offsetted to address anything inside the module. We use hardcoded offsets as they are sufficient for this proof-of-concept. A "*weaponized*" real-world exploit would dynamically determine code layout with the read primitive to work on as many module versions as possible.

chakra.dll. We use the address leak primitive to obtain the address of a JavaScript object. Then we use the arbitrary read primitive twice: first to read the virtual table pointer from the the object, and then to read the address of a function inside `chakra.dll` from the virtual table. We know the offset of this function from the base of `chakra.dll`, so subtracting it from the leaked function address yields the base.

msmpeg2vdec.dll. This module is loaded through various layers of indirection, so a direct leak is difficult. Since we derandomized `chakra.dll`, we know where its import table is located. We read the address of a function imported from `msvcrt.dll` and determine the base of that module. Then, we read the address of a function from `ntdll.dll` from the import table of `msvcrt.dll` and derandomize the former library. Since `ntdll.dll` contains an hash table filled with information about loaded modules, including their base address, we can build a lookup routine on top of the read primitive and get the base of `msmpeg2vdec.dll`.

D. Controlling 64-bit arguments

To apply our attack we need to control a 64-bit argument to an hijacked indirect call. Since Chakra is a C++ application, this would most likely be a virtual call. There are plenty of virtual methods which accept user-controlled 32-bit arguments, such as indexes for string and array operations. However, arbitrary 64-bit arguments are not as easy to come by, particularly because there is no integer 64-bit data type in JavaScript. Many functions accept `Var` arguments, which represent a JavaScript object, either as a pointer to it or as a tagged double, if it is a number [8]. Since we need to set the argument to a pointer to our target, we cannot express it as a number, as it will be tagged. To get a controlled `Var`, we create an array object, which will contain its elements as an array of `Var`s. We then use the write primitive to corrupt one of those elements to the desired value. Since now the element points to code instead of a valid JavaScript object, we have to be careful to not perform operations on it that may crash the engine.

E. Control flow hijacking

We show the general outline of the control flow hijacking stage in Figure 8. We target the `JavascriptFunction::HasInstance` virtual function by hijacking the virtual table pointer for a `JavascriptFunction` object with the write primitive and pointing it to a fake virtual table. Note that virtual tables are read-only, which is why we build a fake table instead of corrupting the real one. We point `HasInstance`, at offset 0x200 in the virtual table, to the S gadget. The function is invoked when the `instanceof` operator is used with a function as the left-hand side operand (step 1). The S gadget (step 2) gets passed a pointer to the `JavascriptFunction` instance object as the first hidden argument, while the right-hand side `Var` is passed as the second one. This is convenient since the virtual call in the S gadget will happen through the same fake virtual table, so by previously setting the entry at 0x50 to point to the PR gadget we can chain it to the S gadget (step 3). We now only need to setup a fake `Var` that points to our target and use it as the right-hand side operand to gain flow control, bypassing CFG.

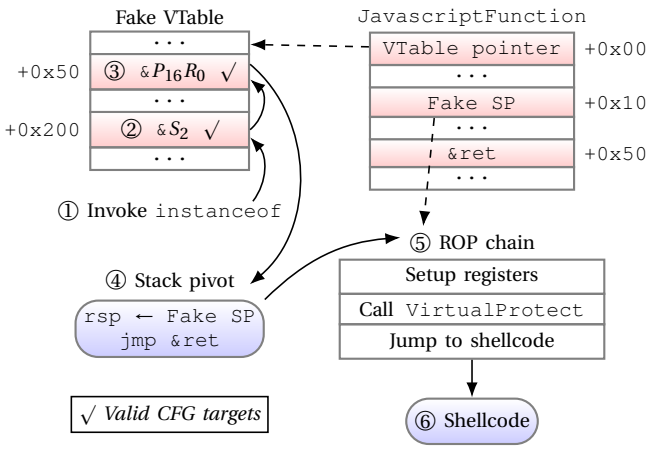


Fig. 8. Overview of control flow hijacking and DEP bypass in our proof-of-concept exploit.

F. DEP bypass

We show an overview of the DEP bypass stage in Figure 8 (steps number 5 and 6 of the exploit). At this point of the attack, we do not have to worry about CFG anymore and we can use standard techniques to bypass DEP. We set up

two strings on the heap: one contains a first-stage ROP chain, the other keeps the second-stage shellcode. We use the address leak primitive to locate those buffers in memory. We redirect control to the stack pivoting gadget shown in Figure 9. The object pointed by `rcx` is again the corrupted `JavascriptFunction`, which we have control over. We use the stack pivoting gagdet to redirect the stack pointer into the ROP chain (step 4). The value that gets loaded into `rdx` is the address of a `ret` instruction to launch the chain. The ROP chain uses gadgets from `chakra.dll` to call `VirtualProtect` on the shellcode buffer to mark it as executable (step 5). Finally, it jumps to the shellcode, achieving arbitrary code execution (step 6).

```
1  ; @ ntdll+0xab305
2  mov rdx, [rcx+0x50]
3  mov rbp, [rcx+0x18]
4  mov rsp, [rcx+0x10]
5  jmp rdx
```

Fig. 9. The stack pivoting gadget from `ntdll.dll` used in our proof-of-concept exploit.

VIII. Discussion and countermeasures

We believe BATE is a real threat. Apart from the standard requirement to be able to disclose the memory layout, we only require control of an argument to an hijacked, CFG-protected indirect call. This a realistic assumption: for example, if the attack is based on virtual table hijacking, the adversary can hijack any function within a virtual table with no extra effort, since a fake table has to be already in place. This allows to freely choose between a large number of candidate functions: it is likely that one will satisfy the requirements. On 32-bit systems BATE is easy to carry out and PR gadgets are widespread, especially in the C/C++ runtime library, effectively making it an almost universal CFG bypass. 64-bit exploitation presents further challenges, such as controlling the indirect call at the end of an S gadget. However, we think BATE is still feasible, as other attacks against CFI by chaining gadgets via hijacked indirect branches [17], [34] proved to be feasible in the past and we demonstrated our attack on real-world code.

A. Countermeasures

We believe that a widely deployable countermeasure needs to modify CFG as little as possible, and should not alter its core design. At its core, BATE relies on a CFG's design assumption being often violated by compiled code. Specifically, CFG guarantees single-byte granularity only for 16-byte aligned targets; however, the compiler sometimes does not properly align functions. There are two ways to address this issue: (i) by improving CFG's precision, or (ii) by avoiding unaligned targets. With the current design, (i) would require at least a bit for every address, resulting in a bitmap that occupies 1/8th of the process' addressing space. While a similar memory footprint is present in previous work [28], and most memory would be virtual and not physically backed, it is still a big price to pay, especially on 32-bit where the virtual address space is limited. Option (ii) is simpler and more feasible: the compiler should align all CFG targets to a 16-byte boundary by inserting appropriate padding. As shown in Table I, 64-bit libraries contain relatively few unaligned targets, so this should not result in a significant increase in code size. On 32-bit, where there are many more unaligned targets, padding could take up a significant amount of space, and possibly have performance implications (e.g., excessive padding could hinder the effectiveness of caches). Despite such downsides, we believe the latter could be the best "immediate" mitigation to what is essentially a design decision, coming from the delicate tradeoff between precision and performance required by CFI techniques.

An additional angle for defense stems from another assumption BATE makes: lack of backward-edge CFI. A shadow stack [12] would protect the return address and stop BATE. Shadowing return addresses was attempted by Microsoft with the Return Flow Guard mitigation, although it was ultimately removed because it suffered from a design-level bypass [44]. A novel and promising hardware-based implementation of shadow stack is Intel CET [19].

IX. Conclusions

In this paper, we presented Back To The Epilogue (BATE), a novel bypass for Microsoft's Control Flow Guard (CFG). After describing the internals of CFG and discussing its weaknesses, we defined two kinds of gadgets, PR and S. PR gadgets can be found near the beginning of unaligned functions. Because of how CFG approximates valid call targets, these gadgets are considered unintended valid targets as well. We then combined S and PR gadgets to implement BATE and bypass CFG. Our technique hijacks control flow to an arbitrary location, thus completely bypassing integrity checks, and allows to launch more traditional exploitation methods.

We assessed the availability of our gadgets to understand the feasibility and impact of BATE: we ran a complete assessment of Windows 10 system libraries, and found many occurrences of our gadgets, even in appealing targets (such as C/C++ runtime libraries, the JavaScript engine, a media codec, and Microsoft Office). Every application that loads any library that contains our gadgets is exposed to BATE. We therefore conclude that BATE is a realistic threat, both on 32-bit and 64-bit systems. We demonstrated this by using BATE to build a remote code execution exploit against the Microsoft Edge browser, a high-value target (because it can be exploited remotely, as the victim only needs to visit, for example, a compromised webpage).

BATE is a so-called "mitigation bypass", because it avoids a security mechanism, in particular by leveraging a design tradeoff between security and memory cost. Therefore, countermeasures are not easy to implement: CFG can hardly be modified to increase its precision. We proposed some possible mitigations to Microsoft, together with our responsible disclosure of BATE. We think the most feasible countermeasure, in the short term, is to force alignment of unaligned targets. However, cost of this approach is unclear, and requires further analysis: padding could take up a significant amount of space, and break caching optimizations. Other proposed future work is to further analyze

Microsoft's CFG, to understand if there are other types of exploitable gadgets next to unaligned targets, and to harden its design by using secondary protection mechanisms that could detect such unwanted valid targets, and enforce calls only of intended targets.

ACKNOWLEDGMENT

Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), by the project CNR-MOST/Taiwan 2016-17 "Verifiable Data Structure Streaming", the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation, and by the grant "Scalable IoT Management and Key security aspects in 5G systems" from Intel.

REFERENCES

[1] "Capstone." [Online]. Available: https://www.capstone-engine.org/

[2] "CVE-2016-7200." [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7200

[3] "CVE-2016-7201." [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7201

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM TISSEC*, 2009.

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ACM ASIACCS*, 2011.

[6] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *IEEE S&P*, 2016.

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2017.

[8] A. Chatra, "Tagged float." [Online]. Available: https://abchatra.github.io/TaggedFloat/

[9] Y. Chen, "The birth of a complete IE11 exploit under the new exploit mitigations," in *SyScan Singapore*, 2015.

[10] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM CCS*, 2015.

[11] D. Dai Zovi, "Practical return-oriented programming," in *SOURCE Boston*, 2010.

[12] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ACM ASIACCS*, 2015.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security 14*, 2014.

[14] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM CCS*, 2015.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *IEEE/ACM MICRO*, 2016.

[16] F. Falcón, "Exploiting CVE-2015-0311, part II: Bypassing control flow guard on Windows 8.1 Update 3," 2015. [Online]. Available: https://www.coresecurity.com/blog/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3

[17] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE S&P*, 2014.

[18] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.

[19] Intel, "Control-flow enforcement technology preview." [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[20] H. Li, "Control flow guard improvements in Windows 10 Anniversary Update," 2016. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/control-flow-guard-improvements-windows-10-anniversary-update/

[21] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying," in *NDSS*, 2017.

[22] Microsoft, "ChakraCore." [Online]. Available: https://github.com/Microsoft/ChakraCore

[23] ——, "Control Flow Guard." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[24] ——, "Argument passing and naming conventions," 2016. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions

[25] ——, "Overview of x64 calling conventions," 2016. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/overview-of-x64-calling-conventions

[26] ——, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003," 2017. [Online]. Available: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

[27] MJ0011, "Windows 10 control flow guard internals," 2014. [Online]. Available: http://www.powerofcommunity.net/poc2014/mj0011.pdf

[28] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *ACM CCS*, 2013.

[29] PaX Team, "Address space layout randomization (ASLR)," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[30] G. Ramalingam, "The undecidability of aliasing," *ACM TOPLAS*, vol. 16, no. 5, pp. 1467–1471, 1994.

[31] rix, "Smashing C++ vptrs," *Phrack Magazine*, vol. 56, no. 8, 2000. [Online]. Available: http://phrack.org/issues/56/8.html#article

[32] M. Schenk, "Back to basics or bypassing Control Flow Guard with Structured Exception Handler." [Online]. Available: https://improsec.com/blog//back-to-basics-or-bypassing-control-flow-guard-with-structured-exception-handler

[33] ——, "Bypassing Control Flow Guard in Windows 10." [Online]. Available: https://improsec.com/blog//bypassing-control-flow-guard-in-windows-10

[34] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *IEEE S&P*, 2015.

[35] F. J. Serna, "The info leak era on software exploitation," in *Black Hat USA*, 2012.

[36] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM CCS*, 2007.

[37] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE S&P*, 2016.

[38] A. Sintsov, "Jit-spray attacks & advanced shellcode," in *HITBSecConf Amsterdam*, 2010.

[39] K. Sun, Y. Ou, Y. Zhao, X. Song, and X. Li, "Never let your guard down: Finding unguarded gates to bypass control flow guard with big data," in *Black Hat Asia*, 2017.

[40] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE S&P*, 2013.

[41] J. Tang, "Exploring control flow guard in Windows 10," 2015. [Online]. Available: https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

[42] Theori, "chakra.dll info leak + type confusion for RCE." [Online]. Available: https://github.com/theori-io/chakra-2016-11

[43] 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 14*, 2014.

[44] D. Weston and M. Miller, "Microsoft's strategy and technology improvements toward mitigating arbitrary native code execution," in *CanSecWest 2017*.

[45] R. Wojtczuk, "An interesting detail about control flow guard," 2015. [Online]. Available: https://blogs.bromium.com/an-interesting-detail-about-control-flow-guard/

[46] P. Yosifovich, A. Ionescu, and D. A. Solomon, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th ed. Microsoft Press, 2017.

[47] Y. Yu, "Bypass DEP and CFG using JIT compiler in Chakra engine." [Online]. Available: http://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/

[48] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[49] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE S&P*, 2013.

[50] Y. Zhang, "Bypass control flow guard comprehensively," in *Black Hat USA*, 2015.

APPENDIX A
S GADGETS

Table III shows detailed information about S gadgets in system libraries. Gadgets of a library are visually separated with a "•". Each gadget is described as a set of spills, separated by commas, in the form `reg@+off`, where `reg` is a CPU register and `off` a stack offset within the RPA. When the tail jump is reached, the value `reg` had upon entry in the S gadget is at `rsp+off`. As such, chaining a $P_{off}R_r$ gadget will hijack the instruction pointer to the entry value of `reg`.

TABLE III. DETAIL OF S GADGETS FOUND IN WINDOWS 10 64-BIT SYSTEM LIBRARIES. GADGETS ARE SEPARATED BY "•"

Library	Total S gadgets	S gadgets (deduplicated)
aadtb.dll	3	rbx@+8
Chakra.dll	52	rcx@+8 • rcx@+8, r8@+24 • rcx@+8, rdx@+16 • rcx@+8, rdx@+16, r8@+24
Chakradiag.dll	1	rcx@+8, rdx@+16
CoreUIComponents.dll	1	rdx@+8
d2d1.dll	1	rdx@+8
d3d10warp.dll	1	rbx@+8
D3DCompiler_47.dll	64	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
dbghelp.dll	76	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
edgehtml.dll	76	rcx@+8 • rcx@+8, r8@+24 • rcx@+8, rdx@+16 • rcx@+8, rdx@+16, r8@+24
FlashUtil_ActiveX.dll	2	rbx@+8
jscript9.dll	34	rcx@+8 • rcx@+8, rdx@+16 • rcx@+8, rdx@+16, r8@+24
jscript9diag.dll	5	rcx@+8, r8@+24 • rcx@+8, rdx@+16 • rcx@+8, rdx@+16, r8@+24
mrt_map.dll	3	rbx@+8, rbp@+16, rsi@+24, rdi@+32
mshtml.dll	217	rcx@+8 • rcx@+8, xmm1@+16 • rcx@+8, r8@+24 • rcx@+8, rdx@+16 • rcx@+8, rdx@+16, r8@+24 • rcx@+8, rdx@+16, r8@+24, r9@+32
msvcp120_clr0400.dll	41	rbx@+8 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
msvcr120_clr0400.dll	12	rbx@+8
ortcengine.dll	28	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16
pdm.dll	24	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16
pidgenx.dll	2	rbx@+8, rbp@+16, rsi@+24
rgb9rast.dll	4	rbx@+8 • rbx@+8, rsi@+16
rometadata.dll	3	rbx@+8 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
rtmcodecs.dll	12	rbx@+8 • rbx@+8, rsi@+16
rtmmvrortc.dll	2	rbx@+8
rtmpal.dll	83	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
rtmpltfm.dll	129	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16 • rbx@+8, rsi@+16, rdi@+24
sppc.dll	6	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rsi@+16
sppcext.dll	1	rbx@+8, rsi@+16
SystemSettings.Handlers.dll	7	rbx@+8
SystemSettingsThresholdAdminFlowUI.dll	12	rbx@+8 • rbx@+8, rbp@+16, rsi@+24, rdi@+32 • rbx@+8, rsi@+16
Windows.Media.Protection.PlayReady.dll	20	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rsi@+16
Windows.UI.Input.Inking.Analysis.dll	58	rbx@+8 • rbx@+8, rbp@+16, rsi@+24 • rbx@+8, rsi@+16
WsmSvc.dll	5	r8@+24, r9@+32 • r9@+32