Out Of Control: Overcoming Control-Flow Integrity

Enes Göktaş*
Vrije Universiteit
Amsterdam, The Netherlands
Email: enes.goktas@vu.nl

Elias Athanasopoulos†
FORTH-ICS
Heraklion, Crete, Greece
Email: elathan@ics.forth.gr

Herbert Bos
Vrije Universiteit
Amsterdam, The Netherlands
Email: herbertb@cs.vu.nl

Georgios Portokalidis
Stevens Institute of Technology
Hoboken, NJ, USA
Email: gportoka@stevens.edu

***Abstract*—As existing defenses like ASLR, DEP, and stack cookies are not sufficient to stop determined attackers from exploiting our software, interest in Control Flow Integrity (CFI) is growing. In its ideal form, CFI prevents flows of control that were not intended by the original program, effectively putting a stop to exploitation based on return oriented programming (and many other attacks besides). Two main problems have prevented CFI from being deployed in practice. First, many CFI implementations require source code or debug information that is typically not available for commercial software. Second, in its ideal form, the technique is very expensive. It is for this reason that current research efforts focus on making CFI fast and practical. Specifically, much of the work on practical CFI is applicable to binaries, and improves performance by enforcing a *looser* notion of control flow integrity. In this paper, we examine the security implications of such looser notions of CFI: are they still able to prevent code reuse attacks, and if not, how hard is it to bypass its protection? Specifically, we show that with two new types of gadgets, return oriented programming is still possible. We assess the availability of our gadget sets, and demonstrate the practicality of these results with a practical exploit against Internet Explorer that bypasses modern CFI implementations.**

Keywords*—*Control-flow integrity evaluation, code-reuse attack

I. Introduction

Since the broader adoption of a variety of protection mechanisms, exploiting software vulnerabilities has become more challenging [1]. In particular, the introduction of hardware support for non-executable data regions and its ensuing support from operating systems (e.g., data execution prevention or DEP [2]), the incorporation of stack smashing protection (SSP) [3] in compilers, and the use of address-space layout randomization (ASLR) [4] by applications have significantly raised the bar for exploit writers. However, the sophistication level of attackers has also risen. Information leakage and guessing attacks [5], [6] enable attackers to construct exploits [1], [7], [8] that bypass ASLR and SSP, and are now commonly part of the exploit. Even the smallest initial leak of a code pointer can help expose significant portions of the program in memory [5], while code reuse attacks employing return-oriented programming (ROP) [9], [10], jump-oriented programming (JOP) [11], [12], and return-to-libc [13] are used to overcome DEP.

None of the attacks mentioned above would be possible, if we can prevent an exploit from hijacking the control flow of a program. Enforcing control-flow integrity (CFI), as proposed by Abadi et al. [14], guards against flows of control not intended by the original program. Restricting program flow in this manner prevents all such attacks. Additionally, CFI is not vulnerable to information leakage, but it assumes that DEP is in place. Since its inception in 2005, there have been tens of follow-up publications [15], [16], [17], [18], while it has been particularly successful on other domains like software-fault isolation [19], [20].

Two main problems have prevented CFI from being deployed in practice. First, many CFI implementations require source code or debug information that is typically not available for commercial software. Second, the technique is expensive with overheads that may be as high as 25% [20] to 50% [14]. It is for this reason that much of the recent research focuses on making it fast and practical. Specifically, recent work on practical CFI is applicable on binaries, but it enforces a *looser* notion of control flow integrity [17], [16].

It is crucial to question what the implications of such looser notions of CFI are for the security of the system. Can they still prevent code-reuse attacks, like return-to-libc and ROP? If not, how hard are they to bypass? Because of the importance of CFI and its potential benefits, it is imperative that we can answer these questions.

This work provides an answer to the above questions by evaluating the effectiveness of state-of-the-art CFI solutions. We examine three versions of CFI: the original implementation described by Abadi et al. [14], and recent works by Chao Zhang et al. [16] (CCFIR) and Mingwei Zhang et al. [17] (bin-CFI). We identify the control-flow restrictions they impose, and compose a *conservative* model of the most restrictive framework, which is CCFIR. We then proceed to develop a methodology for constructing code-reuse attacks under this most-restrictive model. Our approach follows the steps of an attacker, who would target a CFI-protected system. First, we identify permissible control-flow transfers. We then proceed to identify and collect various types of code-segments, or gadgets in ROP terminology, that can be invoked given the allowable control-flow transfers. After, we link these segments, frequently in a similar fashion to conventional code-reuse attacks. Finally, we use a chain of our CFI-resistant gadgets to *inject and execute shellcode*.

We demonstrate the feasibility and effectiveness of our methodology by constructing a proof-of-concept exploit, based on a real-world exploit [21] against Internet Explorer 8 on Windows 7 with DEP and ASLR in use.[1] Briefly, the exploit is based on a heap overflow that serves a dual purpose, so unlike other attacks, we only use a single vulnerability for the whole attack. First, it serves as a memory disclosure bug, which we use to locate the addresses of loaded components

[1]The exploit used was a winner in the 2012 Pwn2Own competition http://pwn2own.zerodayinitiative.com/.

and of the gadgets we need to perform the attack. Second, it grants us control over a jump instruction. We redirect the controlled jump to a chain of gadgets linked through call instructions to corrupt the stack and, eventually, control a return instruction. We proceed to perform stack-pivoting [10] to be able to perform ROP using the gadgets available to us. Next, we use a chain of gadgets, linked through return instructions, to make a code segment writable and overwrite it with our shellcode. The final step executes one last, CFI-checked, but permissible, return instruction into our shellcode.

Contributions: In this paper, we:

- evaluate fast, state-of-the-art CFI techniques and show that they do not protect against advanced ROP exploits;
- develop a methodology for performing code-reuse attacks against CFI-protected software;
- we demonstrate the chaining of gadgets using function calls to perform useful actions, i.e., call-oriented programming (COP);
- construct a working exploit for Internet Explorer 8 on Windows 7 with DEP and ASLR on, and assuming CCFIR is in place;
- assess the availability of gadgets required to launch such attacks against other popular software on Windows 7, such as Internet Explorer 9, Acrobat Reader XI, the Microsoft Office 2013 suite of programs, and Firefox 24.

Although we show that current CFI proposals are still vulnerable to ROP exploits, we do not dispute that they (again) raise the bar for attackers. Instead, we propose that CFI is augmented with a run-time mechanism for enforcing integrity. In their original CFI work [14], Abadi et al. had already suggested the use of a run-time mechanism for checking that functions return to their caller [22]. Even though this approach also suffers from applicability problems, due to the counter-intuitive asymmetry between call and return instructions in binaries, we want to reassert the need for run-time defenses to complement CFI toward comprehensive protection from control hijacking and code-reuse attacks. It is our hope that this work will encourage additional research in the direction of CFI to further improve the provided security guarantees and performance.

The rest of this paper is organized as follows. Section II provides some background information on CFI. It compares an ideal version of it with recent more practical instantiations, and presents their limitations. Section III presents our methodology for performing code-reuse attacks under CFI. Based on the steps presented in Sec. III, we describe how we built a working attack against Internet Explorer in Sec. IV. In Sec. V, we analyze different applications on Windows 7 and show that our attack is generic and applicable on many other applications, as its components are found in abundance in many different libraries and programs. Section VI discusses other defenses that may be vulnerable to our attack and identifies future directions for defending against it, and code-reuse attacks in general. Related work, even though discussed throughout the paper, is in Sec. VII. We conclude the paper in Sec. VIII.

II. Control-Flow Integrity

In this section we present background information on control-flow integrity (CFI). We begin by presenting an ideal version of it, we expand on how it is practically implemented, and we conclude by summarizing the weaknesses of its different instantiations.

A. Ideal CFI

The vast majority of control-flow hijacking attacks operate by exploiting a memory corruption bug, such as a buffer overflow, to control an indirect control-flow transfer instruction in the vulnerable program, most commonly function pointers and return addresses. Control hijacking attacks lead to code-reuse (e.g., return-to-libc [13], ROP [9], and JOP [11], [12]) and code-injection attacks.

CFI thwarts control-hijacking attacks by ensuring that the control flow remains within the control-flow graph (CFG) intended by the programmer. Every instruction that is the target of a legitimate control-flow transfer is assigned a unique identifier (ID), and checks are inserted before control-flow instructions to ensure that only valid targets are allowed. All programs usually contain two types of control-flow transfers: *direct* and *indirect*. Direct transfers have a fixed target and they do not require any enforcement checks. However, indirect transfers, like function calls and returns, and indirect jumps, take a dynamic target address as argument. As the target address could be controlled by an attacker due to a vulnerability, CFI checks to ensure that its ID matches the list of known and allowable target IDs of the instruction. An example is shown in Fig. 1. Note that this CFI is *even stricter* than the original proposal [14].

CFI, along with W$\oplus$X protection such as Data Execution Prevention (DEP) [2], and ASLR provides strong guarantees regarding the integrity of the protected programs. *However, there are two major challenges for the adoption of CFI in its ideal form*. First, it requires a complete and precise CFG of the protected application in order to accurately identify all indirect transfer targets and assign IDs. A poor identification of IDs would result in breaking applications. Second, it incurs a non-negligible performance overhead, caused by the introduced checks before indirect control-flow instructions. The larger the number of possible legitimate targets an instruction has, the higher the number of the checks required and the overhead.

B. Practical CFI

We can improve CFI's performance by reducing the number of IDs (also referred to as labels) used in a program. Such a reduction also simplifies the ID checks required before an indirect transfer. For instance, in Fig. 1, we can replace labels `ID41` and `ID51` by a shared label `ID40`, so that we need to check only for the new label before *call *(fptr)* in function *sort*. Note that if we are just examining the code snippets in Fig. 1, collapsing the two IDs into one does not compromise the CFG being enforced by CFI. Similarly, we can replace IDs `ID11` and `ID21` with a new label `ID10` that will be checked before *sort* returns.

Perhaps the biggest challenge in applying the ideal form of CFI is obtaining the complete CFG for a program, particularly

```
bool less_than(int x, int y);          bool greater_than(int x, int y);
bool sort(int a[], int len, comp_func_t fptr)
{
   ...
      if (fptr(a[i], a[i+i]))
         ...
   ...
}
void sort_1(int a[], int len)          void sort_2(int a[], int len)
{                                      {
   ...                                    ...
   sort(a, len, less_than);               sort(a, len, greater_than);
   ...                                    ...
}                                      }
```

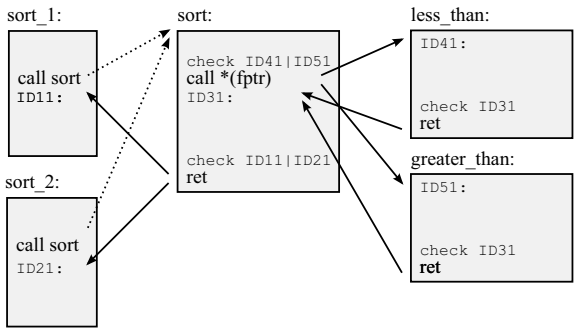


Fig. 1: An example program and its CFG. Conceptually, CFI introduces labels and checks for all indirect transfers. Control-flow transfers checked by CFI are shown in solid lines.

when source code or debugging symbols are not available. Like pointer analysis [23], perfect CFG generation is an intractable problem in many setups. For example, if a function other than *less_than* and *greater_than* is passed to *sort*, and we fail to determine this control flow during the analysis of the program, then the check before *call *(fptr)* will fail, terminating the application.

To tackle the problem of incomplete CFGs, the original CFI proposal, as well as more recent work like CCFIR and bin-CFI, adopts more relaxed rules when enforcing control flow. The approach they take is to coalesce multiple control-flow-target IDs into a single one—essentially permitting more control-flow transfers than in the ideal case. Considering the challenges of obtaining a precise CFG for most applications, Abadi et al. [14] suggest an implementation of CFI using *only one or two IDs* for all transfer targets. As a result, when using a single ID, indirect transfers through function calls, returns, and indirect jumps are allowed to: (a) functions whose addresses are used in the program (function addresses), and (b) instructions following functions calls (i.e., return addresses). When using two IDs, indirect calls are allowed to (a) and returns to (b).

Recent CFI designs have also made significant progress in applying CFI to binary-only software (CCFIR [16]) and even commercial-off-the-shelf (COTS) software without the need for any debugging or relocation information (bin-CFI [17]). Both of these approaches have proposed improved disassembly techniques to extract the control-flow targets of indirect control-flow instructions in binaries and apply CFI with low performance overhead. In addition, they both use a small number of IDs to restrict the control flow. Bin-CFI uses *two IDs*: one for function returns and jumps, and another one for function calls. CCFIR supports *three IDs*, one for function calls and indirect jumps, one for return addresses in "normal" functions, and a specialized one for return addresses in a set of security-sensitive functions. Table I summarizes the restrictions enforced by the CFI approaches we have discussed. Among the three, bin-CFI requires the least information about the binary being protected, and CCFIR has the strictest rules. We discuss more CFI approaches in Sec. VII.

TABLE I: Allowable transfers in different CFI approaches.

	CFI (1 ID)	**CCFIR** (3 IDs)	**bin-CFI** (2 IDs)
Return addresses	All indirect transfers	All `ret` instructions	`ret` & indirect `jmp` instructions
Return addresses in sensitive functions		`ret` instructions in sensitive functions	
Exported functions		indirect `call` & `jmp` instructions	indirect `call` instructions
Sensitive functions		X	

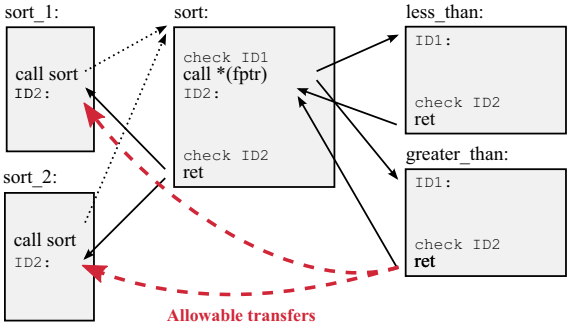
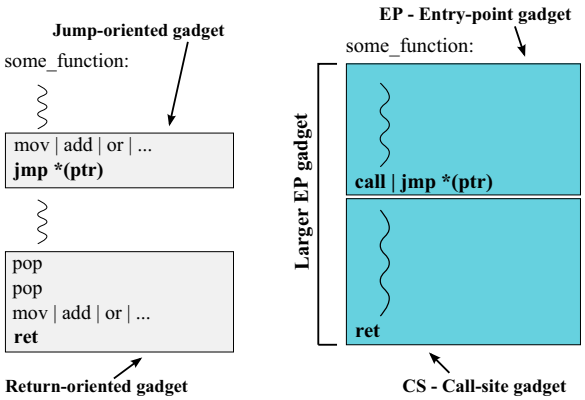


Fig. 2: Loose CFI allows a broader set of control-flow transfers, even though they are not present in the CFG.

C. Weaknesses

CFI restricts control-flow transfers based on a finite, static CFG. As a result, even in its ideal form it cannot guarantee that a function call returns to the call site responsible for the most recent invocation to the function. For example, in Fig. 1 the CFI check before the *ret* in function *sort* cannot enforce that the function is returning to its caller, but only that it returns to one of its legitimate callers in the CFG. In this case, functions *sort_1* and *sort_2*.

Limiting the number of IDs used and applying CFI more loosely to improve performance and accommodate imperfect CFGs, further reduces its precision. Figure 2 shows the CFG from Fig. 1, along with the corresponding CFI checks using only two IDs, much like bin-CFI and CCFIR. The new dashed arrows show control-flow transfers that are not part of the CFG but are still permissible because all return addresses share the



(a) Traditional gadgets. (b) Gadgets under CFI.

Fig. 3: Type of gadgets used in code-reuse attacks.

same CFI ID. Over-permissible transfers are also possible with indirect call and jump instructions. For example, any indirect call could transfer control to *less_than* or *greater_than*.

Undoubtedly, even loose forms of CFI harden binaries against attacks. Normally, control-hijacking exploits are able to redirect execution to any instruction in the binary. On x86 architectures, which use variable-length instructions and have no alignment requirements, an attacker can redirect control to virtually any executable byte of the program. If we consider every executable byte as a potential control-flow target, then CFI blocks more than 98% of these targets [17]. But, *is the remainder 2% enough for attackers exploiting a program?*

It is imperative to question whether loose versions of CFI actually provide sufficient protection against sophisticated exploits [1] developed in recent years. The current state-of-the-art attacks employ ROP to bypass W⊕X protections and memory leaks to bypass ASLR. Attackers carefully develop exploits for a particular application and system configuration to maximize their success rate. The research question in our work is: *can non-ideal versions of CFI withstand such attacks?*

III. Code-reuse Attacks Under CFI

This section presents a methodology for performing code-reuse attacks even when CFI defenses are in place. We model the strictest rules enforced by the CFI mechanisms listed in Tab. I, which are the ones of CCFIR, and show how they can by bypassed. We then proceed to use this methodology to create a working exploit against Internet Explorer 8 in Sec. IV.

A. Traditional Code-reuse Attacks

ROP exploitation is based on an attacker controlling the stack of a program. After corrupting the stack and controlling the return address of an executing function, when the function returns, control is diverted to a gadget specified by the attacker's payload. Since gadgets are small sequences of code that end with a `ret`, similar to the return-oriented gadget shown in Fig.3a, the attacker can carefully position data on the stack to make the program jump from gadget to gadget, chaining together the final code. Between gadget addresses, he can place data to be used in a computation, or arguments for calling a function. Gadgets are usually very simple. For instance, they may pop elements off the stack, perform a small computation like an addition, and then execute `ret`. Such small chunks of functionality form a virtual instruction set for the ROP program that can be chained together to form a program of arbitrary complexity. ROP is an extremely powerful technique that researchers have shown to be Turing complete [9]. Extensions of this technique use gadgets that end with indirect jump or call instructions (e.g., like the jump-oriented gadget in Fig. 3a) [11], [12], [24].

TABLE II: We name gadgets based on their type (prefix), payload (body), and exit instruction (suffix). In total, we name 2×3×3=18 different gadget types.

Gadget type	Payload instructions	Exit instruction
Prefix	Body	Suffix
CS - Call site EP - Entry point	IC - Indirect call F - Fixed function call *none* - Other instructions	R - Return IC - Indirect call IJ - Indirect jump

Creating a working ROP exploit is typically a complex, multi-step process. It typically starts with a memory disclosure that allows the attacker to obtain code pointers. Next, the attack may require a variety of further preparations, such as advanced heap Feng Shui [25] to pave the way for a dangling pointer exploit, stack pivoting, and/or buffer overflows. In addition, the attacker needs to identify useful gadgets and construct a ROP program out of them by setting up the appropriate addresses and arguments on the (possibly new) stack. Finally, a control flow diversion should start off the ROP chain.

The fact that ROP attacks have become so popular despite their complexity demonstrates that attackers will go to great lengths to exploit a system. In addition, they will develop compilers, gadget harvesters, and other tools to make exploitation easier [26], [27], [28]. In other words, it is important that we probe the protection offered by looser forms of CFI very carefully. If we do not, attackers will.

B. Gadgets in CFI

We identify two new types of gadgets that are available to attackers under CFI, shown in Fig. 3b. *Call-site (CS) gadgets* are blocks of instructions right after a call instruction that terminate with a return instruction. *Entry-point (EP) gadgets* are blocks of instructions starting at a function's entry point and ending with an indirect call or jump. Note that there are other types of accessible gadgets, like blocks of instructions beginning after call instructions and ending with an indirect call or jump, but for simplicity we will only focus on the first two types initially.

These new types of gadgets have different properties from traditional gadgets. For instance, they need to begin at an allowable control-transfer pointer. Intuitively, this means that they are going to be larger, both in terms of bytes and instructions. In Sec. V, we collect and measure their length. As the length of the gadgets increases, we need to be more careful when trying to link them together. Primarily, because they will most probably contain instructions unrelated to the ones performing the desired functionality. For instance, suppose we want to load a value from the stack into a register. We still

need to collect the gadgets that have such an effect, but we must also exclude the ones that include instructions that tamper with our data on the stack, or that overwrite the contents of other registers we care about, and so on.

Another consideration is that larger gadgets may include code branches within them. Branches within gadgets can be actually considered both a bane and a blessing. If we cannot control the branching condition, or at least deterministically know its outcome, it provides an additional side effect that needs to be considered before using it. However, if the branch can be controlled, it reveals new possibilities. First, it means that the functional length of the gadget could be smaller, since a part of its instructions can be ignored, revealing a smaller, potentially more valuable gadget. In other cases, it means that *a single gadget can be manipulated to perform different actions by controlling which path it is actually taking*.

Generally, we prioritize gadgets based on their size, examining smaller ones first, as controlling branching conditions is more complex. Table II provides a guide on how we name the gadgets available to us under CFI. We elaborate on their use in the remainder of this section.

C. Locating the Gadgets

Usually the chain of gadgets used in an attack is built offline by manually searching for them or using assisting tools on the vulnerable application and its libraries. The process is straightforward when the layout of the target program in memory is the same on each run. However, ASLR is now becoming more broadly adopted, so predetermining the location of any component in the targeted program is no longer possible.

Attackers have begun using two-stage attacks [29], where they first attempt to learn some information regarding the layout of the target application, and then use that information to fix the location of their gadgets in the payload. This can also become part of a compound exploit that patches itself with the discovered locations [21]. As mentioned earlier, the first stage is usually facilitated by the existence of a memory disclosure bug in the vulnerable application. In their simplest version, such bugs disclose the address of one or more modules loaded in the address space of the victim process, containing gadgets that can be used by the attacker. Bugs of different severity can disclose different amounts of information and of different importance. For example, Snow et al. [5] have shown that even advanced memory inspection is possible, enabling the compilation of an exploit at run time.

Alternatively, previous work has shown that it is also possible to brute-force attack a randomized executable [30], while we have also seen attacks exploiting the fact that one of the modules does not support ASLR [31] (security is only as strong as the weakest link).

The ideal CFI does not require nor is affected by ASLR. However, because in practical versions of CFI the rules are relaxed, ASLR becomes a key component. CCFIR even takes an extra step and proposes an additional level of randomization [16]. For each module, indirect control-flow transfers are redirected through a special springboard section and CFI checks are actually implemented by checking that the target of a transfer is a valid entry in that section. This includes checking for the three different targets (i.e., IDs) supported. Each entry is 8 or 16 bytes long, and their location within the springboard is randomized at load time. As a result, with CCFIR in play an attacker also needs to identify the location of the appropriate entries in the springboard instead of the location of modules. Admittedly, this additional randomization requires more of the attacker, who now would need to exfiltrate both the address of a module, as well as instructions within the module that point to the desired springboard entries.

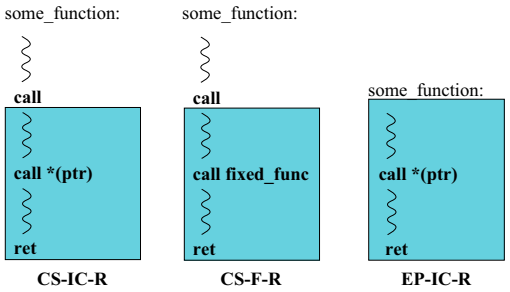


Fig. 4: Common gadgets used for performing function calls. From left-to-right, we show two CS gadgets, the first including an indirect call (IC) and the second a fixed-address (F) call, both ending with a return (R). The last is an entry-point EP gadget containing an IC and ending with a return.

We should emphasize that ASLR has been one of the most important security features introduced in the last decade and has significantly raised the bar for attackers. However, even though we have not seen powerful memory inspection attacks, such as the one described by Snow et al., *yet*, the sophistication of attacks is increasing, and we have witnessed a plethora of real exploits using memory disclosure bugs to bypass ASLR. This paper does not focus on bypassing ASLR. We simply aim to show that given ASLR, DEP, and a loose CFI mechanism code-reuse attacks are still possible, which we demonstrate with a real one-year old exploit in Sec. IV. Therefore, we assume that the attacker has an exploit that grants him the ability to disclose some addresses from the application, and we want to determine, after bypassing ASLR, how hard it is to overcome CFI defenses.

D. Calling Functions

Being able to call a function is crucial functionality for executing an attack. We have the following options under CFI.

1) Through Indirect Calls: Consulting the last row in Tab. I, we notice that indirect call instructions are always allowed to jump to certain functions, i.e., non-sensitive functions that are either exported by a module, or called through a pointer. So, if we control the target of a call instruction, because of the exercised exploit or otherwise, we can immediately call them, launching a return-to-libc attack. The looser the CFI model (e.g., sensitive functions are not distinguished from nonsensitive), the more functions we can call this way.

2) Calling through Gadgets: We can extend the set of functions we can call and how we call them (e.g., also through `ret` and indirect `jmp` instructions) by using special gadgets

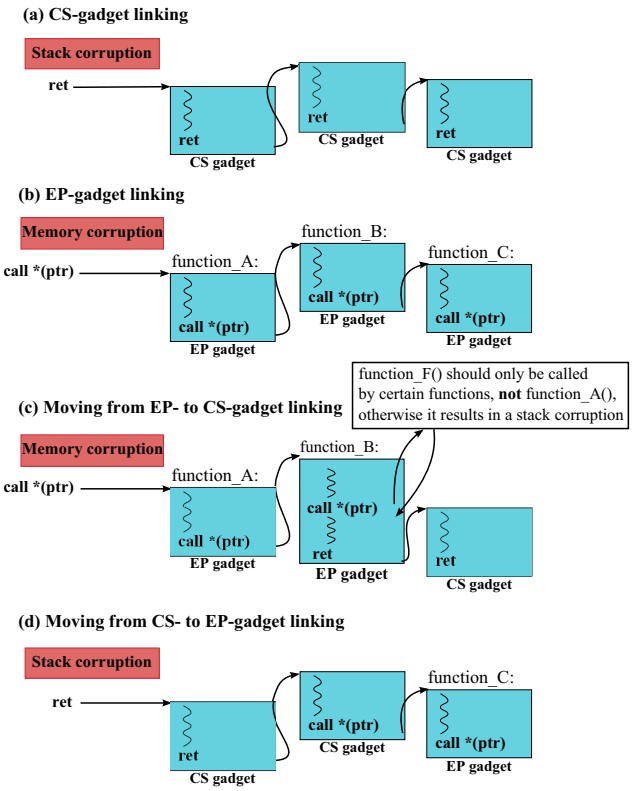


Fig. 5: Different ways of linking gadgets.

like the ones shown in Fig. 4. Essentially, we look for gadgets that include an indirect or fixed function call in their body. For example, redirecting a function return to a CS-IC-R gadget where we control the operand to the indirect call, allows us to use the indirect call within the gadget as discussed above (Sec. III-D1). Most importantly, gadgets that include fixed calls enable us to also call sensitive functions, since there are no CFI checks before direct calls and we can legitimately transfer control to a CS-F-R gadget.

E. Linking Gadgets

We cannot link the gadgets available to us in arbitrary ways, since not all control-flow transfers are permitted (Tab. I). If we control the stack and a return instruction, then we can transfer control to a CS gadget, and set up the stack to chain multiple gadgets of the same type together, as in Fig. 5a. On the other hand, if we first receive control with a call instruction, we can only link EP gadgets, essentially performing call-oriented programming (COP), like in Fig. 5b. Finally, if we control a jump instruction, then based on which CFI model is applied (e.g., CCFIR vs. bin-CFI) we can link CS or EP gadgets.

In certain cases, we may not be able to locate every functionality required in a chain solely composed of CS- or EP-gadgets, and need to link both types of gadgets. Fortunately, there are ways to switch from one gadget type to another. To switch from a chain of EP gadgets to CS gadgets, we actually need to locate an EP-IC-R gadget that calls another function through an indirect call, like in Fig. 5c. We need to carefully select the called and caller functions, so that they make erroneous assumptions with regard to the number of arguments pushed in the stack or the calling convention used (e.g., which one is responsible for popping elements from the stack), with the end goal of corrupting the stack when the called function returns, so we control the next return address. When the EP-IC-R gadget returns, we can redirect control to a CS gadget. The reverse process, shown in Fig. 5d, is more forthright. We link a CS gadget to a CS-IC gadget, where we control the operand of an indirect call. We can use that to redirect control to an EP gadget.

Successfully linking different gadgets depends on our ability to identify the data *used* and *set* by gadgets. We say, a gadget *uses* data in registers or memory when one of its instructions uses the data before it is set. On the other hand, *set data* refers to any data in registers or memory that are set by the gadget before exiting. This is similar to the definitions used in variable liveness analysis [32]. We can use static analysis methods to locate which gadgets can be connected, however this may not be necessary, if the size of the gadgets is relatively small (see Sec. V).

Similarly, calling a function in a gadget chain, depends on our ability to set up its arguments on the stack. We achieve this by linking available gadgets to move arguments from the controlled buffer to the appropriate registers or memory locations. Moreover, we need to find gadgets that preserve at least some function arguments, and in the case of indirect calls the function pointer used. Section IV demonstrates the feasibility of linking available gadgets to perform a successful attack, including calling sensitive functions.

F. From Code-reuse to Code-injection

Traditional code-reuse attacks like ROP and return-to-libc have been shown to be Turing complete [9], [13]. However, with CFI in place fewer gadgets are available to implement any functionality, on any application. Therefore, transforming our code-reuse attack to a code-injection will grant us complete freedom to implement any desired functionality, since CFI applies only to existing code. This strategy is not new. Past exploits [31] begun with a code-reuse payload that sets up a code-injection, bypassing W$\oplus$X semantics.

Such a transformation is achieved by using gadgets to call an API or system call to alter the execute bit on an attacker-controlled buffer and then redirect control to it. This route of action is not possible under CFI because there is no way to directly transfer control to a data buffer. To perform code-injection, we adopt a novel approach that alters the writable bit of existing code, then proceeds to overwrite that code with our shellcode, and finally transfers control to the shellcode, which now resides in a location where CFI will allow an indirect transfer. We adopt such a method to bypass even the strictest of the known CFI mechanisms, CCFIR.

To achieve our goal, we link existing gadgets to first call a function wrapping the system call that allows us to change the permissions of an existing memory area (e.g., *VirtualProtect* on Windows systems). Then, we proceed to copy data from our buffer to the code area using *memcpy*, and finally use an indirect control-flow transfer to invoke our shellcode.

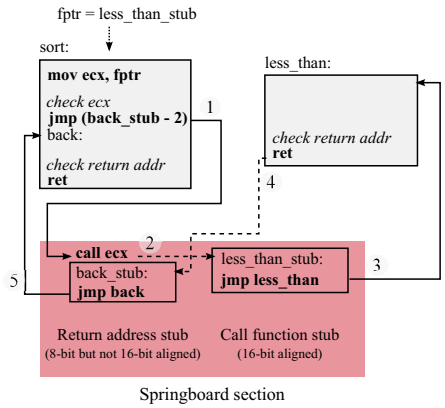


Fig. 6: Function call and return redirection through the springboard section in CCFIR.

IV. Proof-of-Concept Exploitation

In this section, we describe the construction of a proof-of-concept (PoC) exploit, which can compromise a vulnerable binary hardened with CFI. Like before, we assume CCFIR, the strictest of the loose-CFI frameworks, is in place. Nonetheless, the way the exploit works is generic, making it applicable to similar frameworks, like bin-CFI.

The exploit we use is based on a real heap overflow in Internet Explorer [21], which gives control to an indirect jump. The vulnerability is triggered by accessing the *span* and *width* attributes of an HTML table's column through JavaScript. There are some interesting features of this vulnerability, which we would like to point out. First, using this vulnerability we can overwrite a virtual function table (VFT) pointer in a button object, which eventually leads to control over the target of an indirect jump instruction. Second, we can use the same overflow to overwrite a string object's size. We also have a reference to that object, so we can manipulate the string's size to perform a memory disclosure attack. Third, we can trigger the vulnerability multiple times, as long as we are careful so as to not crash the process.

A. Gadget Locations in CCFIR

To better comprehend what data we need to exfiltrate and how gadget chaining is implemented, when CCFIR is deployed, we will briefly explain how it checks indirect control-flow transfers through its springboard section. We refer readers to the original paper [16] for additional details.

CCFIR operates using a special memory layout, where all indirect calls, jumps, and returns are served through a special springboard section. Lets revisit the example from Fig. 2, which we redraw in Fig. 6 to reflect how a function call/return is made in CCFIR. For every function with a relocation entry (CCFIR was implemented for Windows), CCFIR creates a *call function stub*, which is placed in the springboard. Call stubs simply contain a direct jump to their corresponding function. Similarly, every possible return address (i.e., locations following a call instruction) also has a *return address stub* in the springboard. The return stubs include a direct jump back to the corresponding return address, but are also prefixed with an indirect call instruction (*call ecx*).

To secure indirect calls and returns, CCFIR uses the information in the relocation and exported functions section to replace the function pointers in a binary with pointers to their function stubs (e.g., *fptr* ← *less_than_stub*). Notice that this is only done for non-sensitive functions. It then replaces indirect calls with a direct jump to the springboard. The call in the springboard pushes the return stub's address into the stack, so the called function can return safely. Information flow is enforced by aligning the call and return stubs at different address boundaries, emulating this way two IDs, and then checking that the address of a function or a return address follow this alignment. Returns from sensitive functions are omitted for brevity.

B. Exploitation Requirements

To successfully perform the exploitation, we make use of two techniques: heap Feng Shui [25] and heap spraying [33]. The first technique is required to position the vulnerable buffer, string object, and the button object in the right order in the heap, so that when the vulnerable buffer is overflowed the first time, the string object's size property gets overwritten to build the memory disclosure interface. When the vulnerable buffer is overflowed the second time, the button object's VFT pointer is overwritten. The latter technique is required to "spray" a specially crafted buffer in memory. This buffer guides the gadget-chaining process from the initial indirect transfer instruction to code injection. Heap spraying works by making many copies of our buffer to get it allocated at a reliably determined address. This address is written to the VFT pointer of the button object. Even though there is the possibility of failure, heap spraying works very well in this particular exploit, resulting in consistent results over different runs. In the sections below, we will refer to this buffer as the sprayed buffer.

C. Memory Disclosure

The PoC exploit uses memory disclosure for bypassing two constraints. First, we assume ASLR is in place, so we need to resolve the base addresses of modules of interest (e.g., DLLs). Second, because CCFIR uses the springboard to check and authorize indirect transfers, we also need to obtain the addresses of function call and return stubs that correspond to our gadgets. For instance, to return to a CS gadget, we need to use its return address stub, so that the CFI check will not fail.

The vulnerability we use allows us to leak memory by overwriting the size field of a string object and reading values past the original boundary of the object. We now discuss how we leak the base addresses of the two DLLs containing all the gadgets used in the exploit, namely mshtml.dll and ieframe.dll, as well as how we reveal stub addresses in the springboard.

mshtml.dll. The base address of mshtml.dll is revealed by the VFT pointer (not a function pointer) within the button object. The location of this pointer is at a constant relative offset from the string object. Reading this relative offset with the string object, reveals the pointer value. Because this pointer targets a constant known offset in mshtml.dll, the base address of the

library can be derived by subtracting the known offset from the pointer value.

ieframe.dll. To find its base address, we read the address of a function contained in iframe.dll and imported in mshtml.dll. Although, we can compute the absolute address of the location by reading mshtml.dll, we actually need its value relative to the string object. Therefore, the address of the string object has to be revealed first. Fortunately, the button object (positioned after the string object) has a reference to a location that has a fixed relative distance to the string object. By subtracting the fixed relative distance from the reference value, we reveal the string object's address. Once the string object's address is known, the relative distance from the string object to the imported address can be computed. Consequently, the imported address will reveal the base of ieframe.dll.

However, assuming that both these DLLs are protected by CCFIR, the imported function's address that we extract from mshtml.dll actually points to a function call stub in the springboard. To obtain the real location of the function, we need to perform another read to obtain the jump address (or offset) from the stub. The rest of the process is the same as without CCFIR. With the two base addresses in hand, the gadget offsets in the buffer can be located at run time.

CCFIR stubs. Having the modules' base addresses and the offsets for the EP and CS gadgets we intend to use, we need to reveal the stubs corresponding to the gadgets. In a CCFIR-protected library all call sites have direct references to their stubs. Since the call-site offsets in the library are known, direct references to the stubs can be discovered. By resolving these direct references, we get the stub for the call-site gadget. Although the entry-point offsets are also known, they do not reveal their corresponding stub in the springboard. However, the entry points that have a stub in the springboard also have a relocation entry in the code. CCFIR alters the relocation entries such that they point to the corresponding stubs of the entry points in the Springboard. Thus, by resolving the relocation entries, we get the stub for the entry point gadgets.

D. Gadget chaining

The PoC exploit is based on three generic phases. The first phase aims at converting the indirect jump we control to a return instruction. This gives us further flexibility in subverting the control flow of the process, since in the presence of CFI indirect jumps are constrained. However, once we have control over a return instruction, we need to also leverage a fake stack so that we can further execute more gadgets. This task is essentially known as stack pivoting [10] and it is very common in modern exploits, which in our case is a more complicated process.[2] Therefore, we treat it as a separate phase in the exploit's time line. The third and final phases aim at changing memory permissions using available API calls, such as *VirtualProtect* and *memcpy*, for making the shellcode executable and jumping to it. We discuss each of the three phases of the PoC exploit in detail. For each phase, we present the high-level mechanics and (for the interested reader) provide

[2]Stack pivoting usually involves the leveraging of a fake stack by *pointing esp* to the location where the ROP chain is stored. This can be achieved by using just one gadget, which *moves esp* accordingly. Simulating the same task in a constrained environment with EP/CS gadgets is significantly harder.

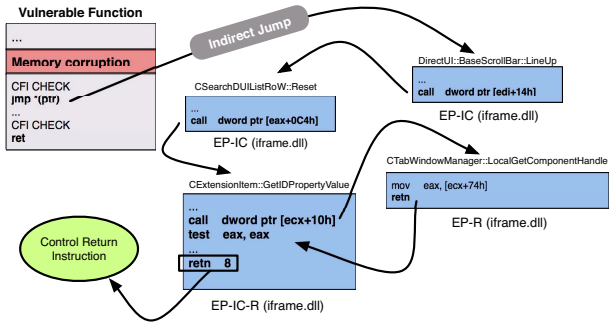


Fig. 7: Schematic representation of how the four EP gadgets are chained in the initial phase of the PoC exploit. Using these four EP gadgets we manage to transfer control from an indirect jump to a return instruction. Notice that we can actually leverage small EP gadgets, which essentially resemble traditional ROP gadgets like the one in Fig. 2 (the EP-R gadget based on the *LocalGetComponentHandle* function) which is composed solely by a *mov* instruction followed by a *retn*.

references to the actual gadget code used, which are listed in the Appendix A.

Phase 1: Transfer control to a return instruction. The exploit we use grants us control of an indirect jump instruction. However, because there are far more CS gadgets in number, as we discuss in Sec. V, and because of the extra steps required to locate EP gadgets, we want to eventually gain arbitrary control of a return instruction. In the case of bin-CFI, jump instructions can transfer control to a CS gadget (Tab. I). However, in the case of CCFIR transferring control from an indirect call or jump instruction to a return instruction is not straightforward. The only function which can be used for this purpose is *longjmp*, which returns to a location that is provided through a buffer. However, CCFIR tags *longjmp* as a sensitive function and prevents any indirect control-flow instruction to direct execution to it.

Therefore, we need to selectively call gadgets that eventually give us control of a return instruction. The main idea is to use the indirect jump we have to jump to code that (a) pushes on the stack a return address which we can influence through the sprayed buffer, and (b) has an indirect call or indirect jump for calling a "stack smasher" or another EP gadget. In our PoC, exploit we have chained four gadgets for carrying out the whole task, which we present in Gadgets 1-1 to 1-4 in Appendix A. Gadget 1-1 pushes an address we control as a return address on the stack and Gadget 1-2 breaks the caller's assumptions by just popping the return address from the stack. Gadget 1-3 sets a pointer to the sprayed buffer as an argument on the stack, but requires control over *edi*, so we use Gadget 1-4 for this. The order in which the gadgets are chained is depicted in Figure 7. Notice that we can actually leverage small EP gadgets which essentially resemble traditional ROP gadget — like Gadget 1-2 which is composed solely of a *mov* instruction followed by a *retn*.

Phase 2: Stack pivoting. The instruction that is executed just before the first controlled return instruction is a *pop* instruction

which loads the top of the stack into *ebp* (see Gadget 1-1). However, the top of the stack contains already a pointer to our buffer and therefore *ebp* is loaded with a value we can influence. Nevertheless, we still have to move the value of *ebp* to *esp*. This is done by having the first controlled return instruction transfer control to Gadget 2-1, which essentially points *esp* to the sprayed buffer. However, we are not there yet, since we want *esp* to point to the ROP chain within the sprayed buffer. We need to further shift *esp* using Gadget 2-2. A familiar reader can notice that chaining these gadgets causes *esp* to increase by 0x30 bytes (12 dwords) and by 0x18 bytes (6 dwords), respectively.

Phase 3: Change memory permissions. So far, in phases 1 and 2, we managed to gain control of a return instruction and point the stack to the ROP chain hosted in the sprayed buffer. The best way to proceed from here is to transform our code-reuse attack to a code-injection attack. Since, conventional ways of injecting code do not work because of DEP and CFI, we adopt a novel approach, which we already discussed in Sec. III-F. Instead of making data executable we overwrite part of the program's code with shellcode. We do so by making a code area writable using a call to *VirtualProtect* and then overwrite it with shellcode using *memcpy*. Since, we cannot directly call *VirtualProtect* because it is considered to be a sensitive function for CCFIR, we use a CS-F-R gadget. *memcpy* on the other hand is not sensitive, so we can also call it using a CS-IC-R gadget. The two CS gadgets we used are Gadgets 3-1 and 3-2.

V. Evaluation

In this section, our main goal is to show that the type of gadgets we defined in Sec. III and used in Sec. IV are very common and can be found on many popular applications. This serves to show the generality of our attack and to reveal the extent of the problem.

We selected and analyzed six widely used applications on Windows 7 x86: Internet Explorer 9.0.20, Firefox 24.0, Adobe Reader XI 11.0.5, and Word, Excel, and PowerPoint from the Microsoft Office Professional Plus 2013 suite. We launched the binaries and collected the set of libraries each one uses. In total, the whole dataset including applications and libraries consisted of 164 unique PE files. For each of these files, we used the popular disassembler IDA Pro to disassemble and collect the statistics presented throughout this section.

We collected the gadget types described in Sec. III. In particular the following: (EP/CS)-R, (EP/CS)-IC-R, (EP/CS)-F-R, (EP/CS)-IJ, and (EP/CS)-IC. All these types are usable under CFI and we have demonstrated their use in our proof-of-concept exploit. Let us use an example to explain the gadget collection process. Consider the case where we begin disassembling a function within a PE file, and we locate a call instruction. In this case, we define a CS gadget starting directly after that instruction, unless we are at the end of the function. We then analyze the instructions following, until we reach a fixed call to a function that has a resolved function name, or an indirect transfer, such as a return, indirect call, or jump. At that point, we classify that gadget as a CS-F, CS-R, CS-IC, or CS-IJ. For CS-F and CS-IC gadgets, we can continue processing until we find a return, in which case we classify the gadget as a CS-F-R or CS-IC-R instead.

TABLE III: Distribution of gadget types in Internet Explorer 9. For gadgets including branches, we count all paths from a particular CS or EP to an exit instruction.

Gadget Type	Internet Explorer 9 Gadgets w/o Branches	Internet Explorer 9 Gadgets w/ Branches
CS-R	179098	398282
CS-IC-R	12400	124638
CS-F-R	44728	251706
CS-IJ	456	496
CS-IC	59252	297266
Sum	295,934	1,072,388
Unique call sites	295,934	390,910
EP-R	7043	37266
EP-IC-R	2353	7491
EP-F-R	4498	12984
EP-IJ	1183	371
EP-IC	2838	4753
Sum	17,915	62,865
Unique entry points	17,915	27,778

We limit the maximum number of instructions we follow after an entry point to 30 for three reasons. First, the longest gadget our PoC exploit uses is a CS-F-R gadget of 26 instructions. Second, traditional ROP attacks prefer to use shorter gadgets. Finally, we desired to keep the search space for the analysis reasonable.

Gadgets containing branches. During the analysis, we take into account conditional branches within gadgets. In the case of branches, we enumerate distinct paths leading from a gadget entry point (i.e., CS or EP) to an exit point (i.e., R, IJ, or IC), since based on the path followed at run time, a different set (and number) of instructions is executed. So there may be many different gadgets beginning from a specific entry point. These gadgets can be more complex, so we choose to report on them separately from simpler ones that do not include any branches.

Also, for simplicity we exclude gadgets that may be part of a loop. We detect such gadgets by identifying cycles in the static CFG of functions. Note that loops and gadgets with branches can be very useful for attackers, and we do use a few gadgets with branches in the attack described in Sec. IV, but broadly determining their usability requires further research and is beyond the scope of this paper. Table III lists the different types of gadgets found in Internet Explorer 9. We list both gadgets with and without branches. As expected, the number of gadgets including branches is larger than simpler gadgets. It would be, therefore, interesting to investigate ways that these gadgets could be easily used.

Gadget distribution. Table IV lists the number and types of gadgets in various applications. The table lists only simple gadgets that do not include branches. For each application, we first list the total number of gadgets that can be found in the application at run time. This includes the gadgets found in the application binary, as well as *all* the DLLs loaded by the binary at run time.

Calling sensitive functions. Certain system functions are restricted by CFI approaches like CCFIR. To call such functions, like the *VirtualProtect* function used in Sec.IV, we need to use a CS-F-R or EP-F-R gadget. Table V lists the number of such

TABLE IV: Number of gadgets per gadget type per application. For each application we show the numbers in 'all' PE files and in its largest specific PE file. Although ieframe.dll is used also by the Word, Powerpoint, and Excel applications, we added it under IE9 especially because the ieframe.dll is also used in the PoC exploit. Furthermore MSO.dll is the largest specific Microsoft Office PE file and shared by the Word, Powerpoint and Excel applications. The numbers are collected from gadgets without branches.

App	PE file name	Entry Point Gadgets					Call Site Gadgets				
		EP-R	EP-IC-R	EP-F-R	EP-IJ	EP-IC	CS-R	CS-IC-R	CS-F-R	CS-IJ	CS-IC
IE9	all	7043	2353	4498	1183	2838	179098	12400	44728	456	59252
	mshtml.dll	1748	652	126	912	759	38559	5917	4865	52	8638
	ieframe.dll	654	201	363	6	127	18252	1326	5333	31	5684
Adobe Reader XI	all	18303	1772	8447	1082	2085	166175	5641	62500	1480	40091
	AcroRd32.dll	13106	650	3099	778	372	58027	1156	25276	1213	12587
Firefox 24	all	9773	2611	6316	635	2785	233183	8951	52883	465	45949
	xul.dll	3281	1349	1781	172	900	101475	4812	16617	97	23181
Word 2013	all	13955	3764	9563	780	3798	352170	11749	74498	839	129671
	WWLIB.dll	962	413	690	33	284	60289	1665	3426	34	31913
PowerPoint 2013	all	15425	3922	10214	893	3835	351969	10942	90826	987	114909
	PPCore.dll	1842	460	1144	111	236	50625	704	16758	161	12309
Excel 2013	all	14026	3526	9249	837	3638	340511	11320	74226	859	119016
	Excel.exe	1313	214	357	102	198	54414	1274	4005	60	21698
Microsoft Office 2013	MSO.dll	4376	934	2858	124	520	82006	2200	14612	169	24819
Shared	all	2271	731	2682	171	1395	67209	2506	23203	204	15521
	shell32.dll	671	303	690	7	370	21864	1237	10192	54	9127

TABLE V: Number of gadgets that contain fixed calls to sensitive functions (i.e., CS-F-R or EP-F-R) in the largest application-specific PE files. Like in Table IV, ieframe.dll is added under IE9.

App	PE file name	Gadget type	Process (e.g., CreateProcess)	Memory Management (e.g., VirtualAlloc)	Move Memory (e.g., memcpy)	Library Loading (e.g., LoadLibrary)	File Functions (e.g., CreateFile)
IE9	mshtml.dll	EP	0	0	3	0	3
		CS	0	0	173	52	54
	ieframe.dll	EP	0	0	4	0	2
		CS	1	7	51	45	183
Adobe Reader XI	AcroRd32.dll	EP	0	0	1	1	18
		CS	0	10	144	53	65
Firefox 24	xul.dll	EP	0	0	32	0	1
		CS	0	5	407	31	62
Word 2013	WWLIB.dll	EP	0	0	12	0	1
		CS	0	2	87	27	30
PowerPoint 2013	PPCore.dll	EP	0	0	23	0	1
		CS	0	0	40	19	16
Excel 2013	Excel.exe	EP	0	0	9	0	1
		CS	0	4	29	30	6
Microsoft Office 2013	MSO.dll	EP	0	0	28	0	17
		CS	0	12	290	58	284
Shared	shell32.dll	EP	0	0	1	0	4
		CS	1	12	30	12	81

gadgets that can be found in the largest application-specific PE files within our dataset. We observe that the number of such gadgets is significantly smaller from the total number of available gadgets. Nevertheless, functions of particular interest to attackers are accessible. The small number of such gadgets indicates that it may be possible to develop a mitigation technique based on completely eliminating the ability to call sensitive functions through code-reuse.

Size distribution of gadgets. Figures 8 and 9 show the frequency of gadget sizes for gadgets without and with branches respectively in Internet Explorer 9. Surprisingly, we observe that there is a significant number of smaller gadgets, indicating that ROP attacks under CFI are closer to conventional ROP than we thought. Another, interesting observation is that in Fig. 8, there is a peak for gadgets with 21 instructions. After investigating, we determined that this occurs due to a block in the ole32.dll library that has 1021 pointers (JMPs) pointing to it.

VI. Discussion

A. Other Vulnerable Defenses

kBouncer [34] monitors a process to detect function returns to addresses not preceded by a *call* instruction. It leverages Last Record Branch (LBR), a hardware feature recently introduced in the Intel architecture. kBouncer is able to prevent conventional ROP attacks because they use small gadgets and do not attempt to restrict the gadget pool. Gadget chains like the ones described in the paper would evade detection, since they exhibit different behavior.

However, kBouncer also introduces a heuristic based on the observation that ROP attacks perform an excessive number of returns without calls. This heuristic could certainly detect gadget chains consisting entirely of CS-R gadgets. Nevertheless, we believe that we could potentially bypass it by using CS-F-R or CS-IC-R gadgets to call a function simply for tricking kBouncer. More experimentation is required before we can make any confident claims in this area.

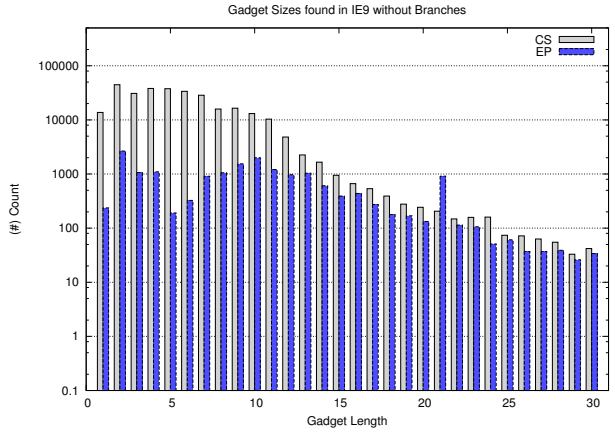


Fig. 8: Frequency of gadgets *without* branches in IE9 based on their length (instruction count).

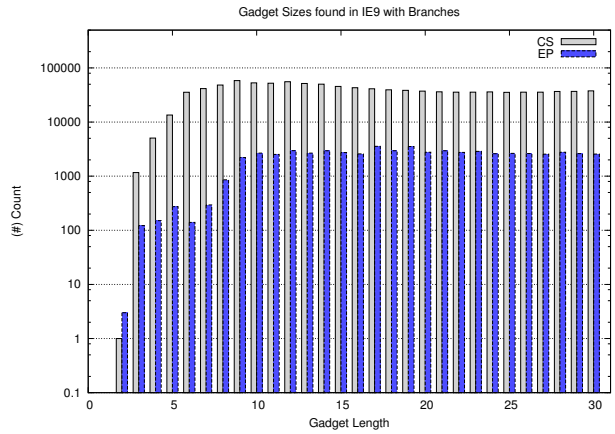


Fig. 9: Frequency of gadgets *including* paths with branches in IE9 based on path length (instruction count).

G-free [35] recompiles a program and performs a set of transformations for eliminating gadgets, enforcing aligned code, and preventing code from jumping in the middle of a function. The latter is enforced by setting a random cookie during function entrance and checking for the cookie's integrity at function exit. This mitigation essentially breaks all CS gadgets we use throughout this paper, since a CS gadget essentially transfers control to a call-site, without entering the function hosting the call site normally. Nevertheless, all EP gadgets can still work and therefore attacks like ret2libc [13] may be still possible, depending on the actual vulnerability. Also, our PoC exploit shows that while chaining EP gadgets is harder, it is possible. Further research is required to determine if EP-F-* gadgets could be also used to bypass G-free.

B. Possible Defenses

In their original CFI work Abadi et al. [14] proposed a shadow call stack [36], [22] that is securely maintained at run time to harden CFI. The shadow stack keeps a copy of the return addresses stored in the program stack, in a way that a prospective attacker cannot overwrite them. For instance, by using segmentation in x86 CPUs, a feature which has been discontinued in 64-bit CPUs. Even though not without problems and incurring additional overhead, such a run-time companion provided higher security guarantees. Unfortunately, correctly tracking a shadow stack is not trivial, as every *call* instruction is not necessarily matched by a *ret*. For instance, compiler-applied optimizations, such as the tail-call optimization, can end a function with a *call*, and the called function is responsible for cleaning up the stack to return directly to the caller's parent. Other optimizations also introduce such asymmetry between the number of calls and returns.

ROPDefender [37] is another approach, also using a shadow stack, that aims to enforce a *call-ret* pairing policy, therefore any *ret* instructions which have not been triggered by a *call* will be detected. However, this approach also suffers from the problems listed above, while it also incurs non-negligible overhead.

Another possible defense is Control-Flow Locking (CFL) [38] which uses locks to preserve CFG integrity. It implements a lock operation before indirect control-flow transfers and an unlock operation before valid targets. The locks and unlocks are paired according to computed key values based on the links in the CFG which is extracted using source code. CFL detects violations once a lock or unlock operation fails. We believe that CFL is a promising direction, and it would be able to prevent our attack, however it is hard to apply in the absence of source code.

A more ad-hoc defense would focus on preventing the application from making its text segment, as well as the springboard in the case of CCFIR, writable. Even though that would not prevent the code-reuse part of our attack, it would raise the bar, as we would no longer be able to inject code.

VII. Related Work

In this section we present related research. We start with a short survey of how defenses have evolved in the face of more elaborate attacks, and then we place our work in context with related CFI research.

A. Advanced Mitigation Techniques

Traditional exploitation through stack smashing and code injection is considered today hard due to the massive adoption of stack canaries [3] and Data Execution Prevention (DEP) [2], respectively, by all of the most popular commodity operating systems. Most of the modern microprocessor architectures also support a special MMU feature, well-known under the generically accepted No eXecute (NX) bit term – although different brands use their own acronym to refer to the same technology – to facilitate DEP. Essentially, DEP will force the process to crash in any attempt to execute code placed in data. Therefore, since foreign code cannot be introduced, adversaries can only utilize existing code in the process image. For example, they can force the program to jump to a particular function after having prepared the stack accordingly. This technique is called return-to-libc [13], reflecting the classic idiom of calling a *libc* function (*system* or the *exec* family)

for spawning a shell. Leveraging existing code for compromising a program has been generalized in Return-Oriented Programming (ROP) [9], where short snippets of code called gadgets are chained together to introduce a new, not initially intended, control flow. ROP can especially take advantage of the density of instruction sets in CISC architectures and the availability of overlapping instructions due to the lack of instruction alignment. Nevertheless, it has been shown that ROP is possible in RISC architectures [27], or it can be carried out without using the *ret* instruction [11], [12] making defenses that are based on the emulation of the *ret* command less effective [24].

Defenses against ROP are based on randomization of a process' layout so that adversaries lack knowledge of where the program's code is mapped in the virtual address space. The most straightforward way to achieve this is by using Address Space Layout Randomization (ASLR) [4], which maps processes and dynamically linked libraries in random addresses each time. Unfortunately, ASLR with low entropy can be brute-forced [30] and, worse, in the presence of memory disclosure, attackers are able to leak a few memory addresses and thus bypass any protection offered by ASLR completely. As a result, many proposals where introduced for fine grained randomization in executables by applying in-place randomization [39], breaking the linearity of the address-space [40], or shuffling the basic code blocks of a program [41]. Again, in the presence of powerful exploits that can arbitrarily leak memory, it has been shown that fine grained randomization techniques fail in protecting the vulnerable program [5]. In this paper, although we use the same powerful exploit [5], we do not take advantage of arbitrarily leaking *all* process memory.

B. CFI Research

Control-Flow Integrity (CFI) [14] was originally introduced as a generic methodology for enforcing the integrity of a program's control-flow graph, as realized by a compiler or discovered using static analysis before program execution, by prohibiting all unseen control flows at run time. Typically, the program's code is marked with labels and checks that validate intended control flows. CFI can be applied in principle in any system, like for example smarthphones [18], [42]. CFI has two major limitations. First, discovering the complete control-flow graph is not always possible – although recent attempts towards that direction are promising [43], [44] – and, second, applying complete CFI in a program often incurs high performance overhead. For this, researchers have attempted to relax CFI by applying it directly to binaries [16], [17]. In this paper, we explore how this relaxation degrades the effectiveness of CFI and how adversaries can take advantage of it for bypassing CFI protection. CFI can be leveraged for enforcing Software Fault Isolation [19] and constraint additional program code in a sandbox. Many popular pure SFI frameworks, like for example Native Client [45], [46] or SFI-inspired policy frameworks, like XFI [47] and WIT [48], employ CFI-checks to prevent flows from *escaping* the sandbox. On the other hand, ideas from SFI implementations can be used for enforcing CFI. For example, CCFIR [16] uses a similar memory layout (Springboard) with the one used by Native Client (NaCl) [45], [46].

VIII. Conclusion

In this paper, we have examined the security implications of looser notions of control flow integrity (CFI). The looser notions of CFI are fast, but allow certain control flows in a program's execution that were not in its original control-flow graph. Specifically, we have shown that such permissible, but incorrect, flows of control allow attackers to launch ROP attacks that by their nature are just as powerful as regular ROP exploits. While the novel ROP chains are based on two new types of gadget (and thus have gadget sets that are more limited than regular ROP), we also show that such gadgets are still widely available and that the gadget set is broad and powerful. Finally, a proof-of-concept exploitation against Internet Explorer, that bypasses modern CFI implementations, demonstrates that our techniques are practical.

As CFI is one of the most powerful defensive measures currently available against advanced exploitation techniques, we believe these results to be highly relevant. Specifically, our results suggest that a CFI solution based on static analysis alone may not be sufficient for comprehensive protection against ROP attacks, and that permitting any additional edges in the control-flow graph introduces vulnerabilities that may be exploited. We expect new CFI solutions to utilize static information as much as possible, but it is unlikely that the stricter notions of CFI can work without the use of run-time information. There is no question that gathering such information comes at a cost in performance, but neglecting to do so comes at the cost of security.

Acknowledgement

We want to express our thanks to anonymous reviewers for valuable comments. This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, or the Air Force. This work was also supported in part by the ERC StG project Rosetta, the FP7-PEOPLE-2010-IOF project XHUNTER, No. 273765, and EU FP7 SysSec, funded by the European Commission under Grant Agreement No. 257007.

References

[1] N. Joly, "Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013)," VUPEN Vulnerability Research Team (VRT) Blog, May 2013, http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php.

[2] S. Andersen and V. Abella, "Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention," Microsoft TechNet Library, September 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, vol. 81, 1998, pp. 346–355.

[4] PaX Team, "Address Space Layout Randomization (ASLR)," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[5] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013.

[6] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the 2nd European Workshop on System Security*, 2009, pp. 1–8.

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

[8] C. Evans, "Exploiting 64-bit Linux like a bos," http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html.

[9] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications security*, October 2007, pp. 552–61.

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

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and Communications Security*, October 2010, pp. 559–72.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ASIACCS*, March 2011, pp. 30–40.

[13] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, 2011, pp. 121–141.

[14] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005, pp. 340–353.

[15] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 380–395.

[16] 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 *Proceedings of the 1013 Security and Privacy Symposium*, 2013, pp. 559–573.

[17] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium*, 2013.

[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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, February 2012.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993, pp. 203–216.

[20] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011, pp. 29–40.

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

[22] Vendicator, "StackShield," http://www.angelfire.com/sk/stackshield/.

[23] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[24] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with return-less kernels," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 195–208.

[25] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[26] Pakt, "Ropc - a turing complete rop compiler," https://github.com/pakt/ropc.

[27] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of CCS 2008*, P. Syverson and S. Jha, Eds. ACM Press, Oct. 2008, pp. 27–38.

[28] Corelan, "Mona: a debugger plugin / exploit development swiss army knife," http://redmine.corelan.be/projects/mona, 2011.

[29] N. Joly, "Technical analysis and advanced exploitation of Adobe Flash 0-day (CVE-2011-0609)," VUPEN Vulnerability Research Team (VRT) Blog, March 2011, http://www.vupen.com/blog/20110326.Technical_Analysis_and_Win7_Exploitation_Adobe_Flash_0Day_CVE-2011-0609.php.

[30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004, pp. 298–307.

[31] N. Joly, "Criminals are getting smarter: Analysis of the Adobe Acrobat/Reader 0-day exploit," VUPEN Vulnerability Research Team (VRT) Blog, September 2009, http://www.vupen.com/blog/20100909.Adobe_Acrobat_Reader_0_Day_Exploit_CVE-2010-2883_Technical_Analysis.php.

[32] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2006.

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

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 447–462.

[35] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 49–58.

[36] M. Frantzen and M. Shuey, "StackGhost: Hardware facilitated stack protection," in *Proceedings of the 10th USENIX Security Symposium*, August 2001, pp. 55–66.

[37] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM ASIACCS*, 2011, pp. 40–51.

[38] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 353–362.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[40] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 571–585.

[41] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[42] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based CFI for iOS," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[43] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *EuroSys*, 2013, pp. 295–308.

[44] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, 2013.

[45] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.

[46] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *Proceedings of the 19th USENIX conference on Security*, 2010.

[47] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 75–88.

[48] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

APPENDIX

A. PoC Exploit Gadgets

```
; library: mshtml.dll
; offset: 0x001BC907

1 mov     ecx, [ecx+24h]
2 mov     eax, [ecx]
3 mov     edx, [eax+24h]
4 jmp     edx
```

Gadget 0-1: (Phase 0, Gadget 1). The few instructions executed before the first controlled indirect control-transfer instruction. First *ecx* gets loaded with a pointer to the button object in the heap (line 1) and then *eax* gets loaded with the object's vtable pointer which is overwritten with a pointer to our buffer by using the heap buffer overflow vulnerability in Internet Explorer 8. Our buffer contains the data required to guide the process in performing a code injection exploit. Because we assume CCFIR is in place, the target of the jmp instruction must be an entry point (see Gadget 1-4).

```
; library: ieframe.dll
; offset:  0x002B6D88
; type:    EP-IC-R

 1 mov     edi, edi
 2 push    ebp
 3 mov     ebp, esp
 4 mov     eax, [ebp+8]
 5 push    dword ptr [ebp+0Ch]
 6 mov     ecx, [eax]
 7 push    dword ptr [eax+0Ch] ; push return address
 8 push    eax
 9 call    dword ptr [ecx+10h]
10 test    eax, eax
11 jge     short loc_76836DA5  ; if eax >= 0:---+
12 mov     eax, 80004005h      ;                |
13 pop     ebp                 ; <--------------+
14 retn    8
```

Gadget 1-1: (Phase 1, Gadget 1). The main gadget used to transfer control from an indirect call to a return instruction. The instruction at line 7 pushes the return address to the stack. If we can point *eax* and the data at *eax+0Ch* to the sprayed buffer, we can make the function return to our desired address. Notice, that *eax* is loaded from the stack (line 4), which we can poison by feeding the right arguments when calling this gadget with an indirect call. In the same fashion, we can call a second gadget by pointing *ecx* in the sprayed buffer through *eax* (line 6). The latter is presented in Gadget 1-2.

```
; library: ieframe.dll
; offset:  0x000A647C
; type:    EP-R

1 mov     eax, [ecx+74h]
2 retn
```

Gadget 1-2: (Phase 1, Gadget 2). This gadget removes only the return address from the stack. By breaking the caller's assumptions, at the moment the caller returns, its return instruction will point to our desired return address, which we have pushed on the stack by means of registers that point to the sprayed buffer (see Gadget 1-1).

```
; library: ieframe.dll
; offset:  0x002869F0
; type:    EP-IC

1 mov     edi, edi
2 push    esi
3 mov     esi, ecx
4 mov     eax, [esi]
5 push    edi                 ; callee's argument
6 call    dword ptr [eax+0C4h]
  ...
```

Gadget 1-3: (Phase 1, Gadget 3). The main gadget (see Gadget 1-1) requires a pointer to the buffer as an argument on the stack so we use this gadget to satisfy this requirement. However, this gadget is based on *edi*, which we have not yet defined. Therefore we use the last gadget of this phase, which is presented in Gadget 1-4.

```
; library: ieframe.dll
; offset:  0x0030B665
; type:    EP-IC

1 mov     edi, edi
2 push    ebp
3 mov     ebp, esp
4 push    ebx
5 push    esi
6 mov     esi, ecx
7 push    edi
8 mov     edi, [esi]          ; edi pointer to buffer
9 call    dword ptr [edi+14h]
  ...
```

Gadget 1-4: (Phase 1, Gadget 4). This gadget defines *edi*, as needed by Gadget 1-3 using *esi*, which is loaded with *ecx* which we already control (see Gadget 0-1).

```
; library: mshtml.dll
; offset:  0x0042E9D9
; type:    CS-R

1 pop     edi
2 pop     esi
3 mov     esp, ebp
4 pop     ebp
5 retn    20h
```

Gadget 2-1: (Phase 2, Gadget 1). Point *esp* to the sprayed buffer.

```
; library: mshtml.dll
; offset:  0x004A73E3
; type:    CS-R

1 retn    14h
```

Gadget 2-2: (Phase 2, Gadget 2). Further increase *esp* by 0x18 (6 dwords).

```
; library: ieframe.dll
; offset:  0x001ADCC2
; type:    CS-F-R

 1 push    eax                 ; destination
 2 call    memcpy
 3 add     esp, 0Ch
 4 xor     eax, eax
 5 jmp     short loc_7672DCE7
 6 pop     ebx
 7 pop     edi
 8 pop     esi
 9 pop     ebp
10 retn    8
```

Gadget 3-1: (Phase 3, Gadget 1). Gadget for calling of *memcpy* for copying the shellcode to existing program code.

```
; library: ieframe.dll
; offset:  0x0006FBAE
; type:    CS-F-R

 1 and     dword ptr [ebp-0Ch], 0
 2 lea     eax, [ebp-0Ch]
 3 push    eax          ; address to save old protection
 4 push    40h              ; new protection
 5 push    ebx              ; size
 6 mov     ebx, [ebp-8]
 7 push    ebx              ; address
 8 call    ds:VirtualProtect
 9 test    eax, eax
10 jz      loc_766E9531 ; if eax==0: goto error handler
11 mov     eax, [ebp+8]
12 and     dword ptr [edi+4], 0
13 mov     [edi+8], eax
14 mov     [edi+10h], esi
15 mov     [edi+0Ch], ebx
16 mov     eax, [ebp-0Ch]
17 mov     [edi+14h], eax
18 mov     eax, dword_768E2CCC
19 mov     [edi], eax
20 mov     dword_768E2CCC, edi
21 xor     eax, eax
22 pop     edi
23 pop     ebx
24 pop     esi
25 leave              ; == mov esp, ebp and pop ebp
26 retn    14h
```

Gadget 3- 2: (Phase 3, Gadget 2). Gadget for calling *VirtualProtect* and making existing program code writable.