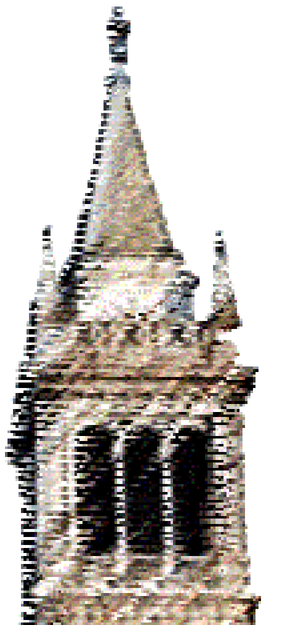
Breaking Active-Set Backward-Edge Control-Flow Integrity



Michael Theodorides

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-78
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-78.html

May 12, 2017

Acknowledgement

I would first like to thank my research advisor Professor David Wagner. David has been a phenomenal advisor and working with David has been a great pleasure. I consider myself very lucky to have had David as an advisor.

I would also like to thank Nicholas Carlini for being a great mentor and including me in his research projects. Nicholas's research first introduced me to many of the topics in this thesis.

I thank Professor Raluca Ada Popa for providing valuable feedback on this thesis and agreeing to be the second reader.

Last but not least, I thank my family for their inspiration and support.

Breaking Active-Set Backward-Edge Control-Flow Integrity

by Michael Theodorides

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor David Wagner
Research Advisor

MAY 5, 2017

(Date)

* * * * * * *

[illegible]

Professor Raluca Ada Popa
Second Reader

May 10, 2017

(Date)

Breaking Active-Set Backward-Edge Control-Flow Integrity

by

Michael Theodorides

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2017

Breaking Active-Set Backward-Edge Control-Flow Integrity

Abstract

Hardware-Assisted Flow Integrity eXtension (HAFIX) was proposed as a defense against code-reuse attacks that exploit backward edges (returns). HAFIX provides fine-grained protection by implementing Active-Set Backward-Edge CFI: confining return addresses to only target call sites in functions active on the call stack. We study whether the active-set backward-edge CFI policy is sufficient to prevent code-reuse exploits on real-world programs. In this thesis, we present five novel attacks that exploit weaknesses in active-set backward-edge CFI and demonstrate these attacks are effective in case studies examining Nginx web server, Exim mail server, and PHP. We then propose improvements to active-set backward-edge CFI that we believe will improve its effectiveness against code-reuse attacks.

To my family

- for their love, support, and encouragement.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my research advisor Professor David Wagner. David has been a phenomenal advisor and working with David has been a great pleasure. I consider myself very lucky to have had David as an advisor.

I would also like to thank Nicholas Carlini for being a great mentor and including me in his research projects. Nicholas's research first introduced me to many of the topics in this thesis.

I thank Professor Raluca Ada Popa for providing valuable feedback on this thesis and agreeing to be the second reader.

Last but not least, I thank my family for their inspiration and support.

Chapter 1

Introduction

Memory-safety vulnerabilities have been used to exploit systems for over two decades. Researchers have studied many defenses against these attacks, yet the performance and other limitations of these defenses have meant that memory-safety exploits remain ubiquitous [18]. Data Execution Prevention (DEP) [25], which marks pages of memory used for data as non-executable, has caused a shift to code-reuse attacks which redirect program flow to code already present in a program [27], instead of code that is injected by an attacker.

Mitigations to code-reuse attacks have included stack canaries [6], address randomization (ASLR) [24], and control-flow integrity (CFI) [1]. Stack canaries detect buffer overflows by confirming that a canary value on the stack frame in between the return address and local variables is unmodified. Stack canaries are vulnerable to attackers guessing and overwriting the random canary value. ASLR rearranges the memory address of the base of the stack, heap, and program's libraries to prevent an attacker from reliably jumping to code of their choosing. ASLR is vulnerable to information disclosure attacks. Stack canaries and ASLR have limitations, and exploits have been demonstrated on both mitigations [29]. Control-flow integrity (CFI) [1] is a promising defense that prevents exploits by confining the execution of a program to a specific control-flow graph (CFG). Any violation in following the CFG raises an exception. However, CFI seems to suffer from a performance/security tradeoff: full-strength CFI imposes a non-trivial performance overhead. Researchers have proposed coarse-grained CFI defenses that reduce the performance overhead by relaxing the security policy [4, 22, 33], but unfortunately these schemes have been demonstrated to be ineffective [12].

Recent research has suggested using hardware to implement CFI. In 2014, Davi et al. proposed an intriguing hardware-assisted approach with a novel policy for restricting return instructions [10]. Their design keeps track of an "active set" of return sites. Each function call adds the subsequent instruction to the set, and return instructions are only allowed to return to an instruction in the active set. In 2015, Davi et al. refined their design, which they dubbed Hardware-Assisted Flow Integrity eXtension (HAFIX) [11]. They also described two hardware implementations of HAFIX, one for the x86 Siskiyou Peak and one for SPARC LEON3, and showed that both implementations achieve excellent performance.

In this thesis, we analyze the security of HAFIX's novel active-set policy for return instructions. This policy provides an interesting intermediate point between coarse-grained CFI and full-strength CFI with a shadow stack. Other researchers have studied coarse-grained CFI (where return instructions are allowed to target any location that follows a call instruction) and fully-precise CFI with a shadow stack (where each return instruction can only return back to the location after the matching call instruction), but the effectiveness of the active-set policy at preventing exploits in real-world programs has not been carefully studied before in the research literature. Our results help understand whether this policy will be sufficient at preventing exploits or if a shadow stack is a requirement for preventing exploits.

To shed light on this question, we examine real-world binaries that had vulnerabilities and evaluate whether HAFIX's active-set policy would have prevented exploitation of those vulnerabilities. We show that this policy can be circumvented when an attacker has write access to arbitrary memory. The active-set backward-edge policy includes many additional execution paths that are not present in CFI with a shadow-stack. These additional paths can lead directly to powerful library calls and as a result enable a number of novel attacks on HAFIX. We present five novel attacks, based on the attacker's ability to return to parent code in a child process after a fork, to earlier call sites in functions on the stack, and to the entry function of a program (typically main). Our results can be viewed as adding to the evidence that a shadow stack is a minimum requirement for CFI, and that weaker policies for return instructions are not sufficient.

To the best of our knowledge we are the first to evaluate the active-set CFI policy on real-world programs. Previous research has speculated about potential weaknesses with HAFIX [5], but the extent that these weaknesses are effective at preventing actual exploits has not been studied.

The remainder of this thesis is structured as follows: in chapter 2 we provide background on code-reuse attacks, CFI, and HAFIX. In chapter 3 we define the threat model. In chapter 4 we describe novel attacks on active-set backward-edge CFI that detail its weaknesses. In chapter 5 we present case studies of real programs with vulnerabilities and demonstrate our attacks on these programs. In chapter 6 we discuss improvements to active-set backward-edge CFI that prevent the attacks in this thesis. Finally, in chapter 7 we conclude the thesis.

Chapter 2

Background and Related Work

2.1 Memory-Related Vulnerabilities and Code-Reuse Attacks

Performance and legacy reasons keep unsafe programming languages such as C and C++ ubiquitous today. These unsafe languages lack overflow checks, automatic memory management, and strong typing and as a result frequently contain errors that lead to memory corruption and exploitable unexpected behavior. Classic attacks work by injecting malicious code through a stack overflow and overwriting the return address to point to the injected code [21]. The widespread adoption of Data-Execution Prevention (DEP) [25] has effectively eliminated code-injection attacks.

Code-reuse attacks have emerged as the replacement to code-injection attacks since the adoption of DEP. Code-reuse attacks reuse fragments of existing program or library code for malicious execution. Leveraging a memory corruption vulnerability to direct control-flow to powerful code already present in the address space of a program has the same effectiveness as injecting code. Code-reuse using libc has been shown to provide turing-complete computation [32].

The most prominent variation of the code-reuse attack is Return-Oriented Programming (ROP) [27]. In ROP small chunks of existing code referred to as gadgets are chained together to achieve malicious activity. Gadgets typically end in a return instruction allowing them to be easily chained together when an attacker has the ability to modify memory and are chosen from the program and its linked library code.

2.2 Code-Reuse Defenses

Defenses for code-reuse attacks have largely fell into two categories: defenses that seek to keep memory safe by combating unsafe programming practices and defenses that assume unsafe programming practices and instead prevent exploitation given vulnerable memory.

Many mitigations have been proposed in the first category [19, 17, 15], however all suffer from performance overhead as large a 67% [20].

A number of defenses have been proposed to mitigate code-reuse attacks given a memory vulnerability. One class of defenses is aimed at randomizing a program's address space to prevent an attacker from diverting control to the specific gadgets they select. Address Space Layout Randomization (ASLR) [24], the leading technique in randomization, rearranges the program's address space by shuffling the addresses of the stack, heap, and libraries. Like most defenses based on address randomization, ASLR is susceptible to information disclosure attacks [28, 29]. Stack canaries [6] are another widely deployed defense that write a canary value in the stack and verify that canary value is present later in the execution to ensure a buffer has not overflowed the stack. Stack canaries prevent the continuous overwriting of the stack but are ineffective at preventing a discrete overwrite [29].

2.3 Control-Flow Integrity (CFI)

Control-Flow Integrity (CFI) [1] is a code-reuse defense that confines a program's execution to be consistent with its static control-flow graph (CFG). The program is monitored at run-time to ensure its control flow follows a valid path in the CFG. Any deviation from the CFG produces an exception.

Ensuring program control-flow stays within a CFG requires validating all indirect control-flow transfers which results in substantial performance impact.

Control transfers can be split into two categories, forward edges (function calls and jumps, including indirect transfers) and backward edges (return instructions). Any CFI implementation must limit both forward and backward edges. Research suggests that at least some forward-edge policies can be enforced efficiently in software [31], but backward-edge policies can be more expensive [9]. This has motivated researchers to examine several different backward-edge policies and to consider hardware support for policy enforcement.

The strongest (most restrictive, most secure) backward-edge policy involves validating return addresses with a shadow stack in protected memory. Each call instruction causes the return address to be pushed to the ordinary stack and to the protected shadow stack; a return instruction validates the return address against the value at the top of the shadow stack. However, shadow stacks impose a significant performance overhead in software, motivating researchers to study weaker, coarse-grained policies for backward edges [16]. One weaker alternative is to omit the shadow stack and check that every return instruction targets the location following some call instruction.

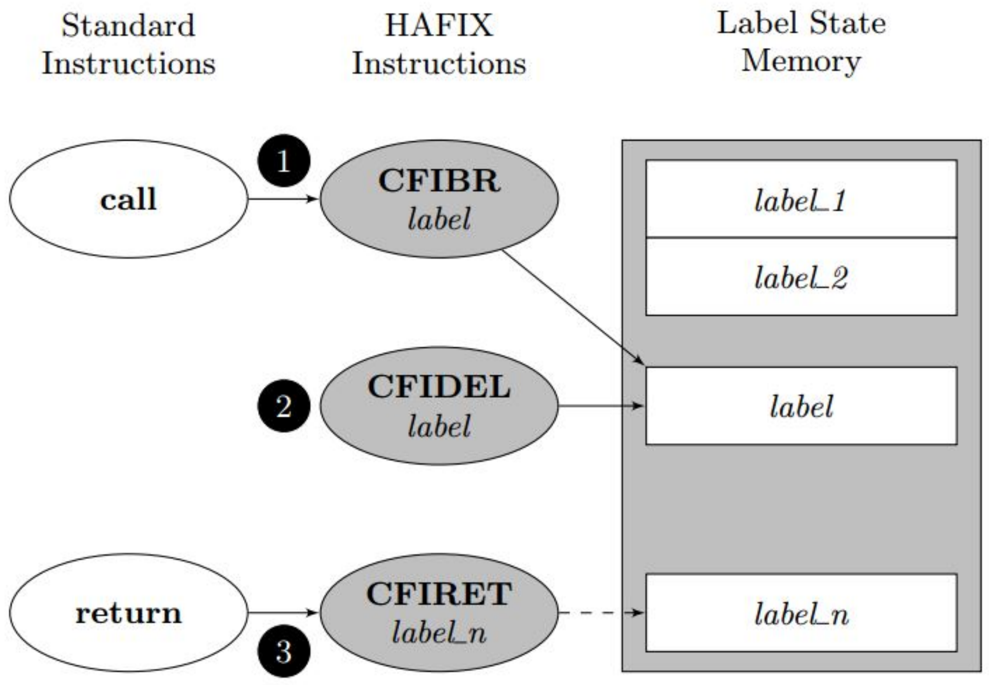


Figure 2.1: The Abstract design of HAFIX CFI. Each function must issue a CFIBR instruction when called to load the label of the function into label state memory (or the active-set). Label State Memory (the active-set) indicates functions that are active. CFIDEL instructions deactivate active functions by removing labels from label state memory. Return instructions must be followed by a CFIRET instruction that only allows returns to functions in label state memory.

2.4 HAFIX and Active-Set Backward Edge CFI

HAFIX is a hardware CFI implementation for backward edges with a performance overhead of just 2% [11]. Their design [10],[11, § 3] introduces an active-set policy that maintains a set of active functions (functions that are executing on the stack) and restricts returns to only target call-preceded instructions in active functions. This active-set backward-edge CFI policy is used in the HAFIX x86 implementation.

Under HAFIX, the compiler assigns unique labels to each function, then uses the labels in the following three new instructions:

1. **CFIBR**: CFIBR is inserted as the first instruction to each function to insert the function's label into the active set.

2. **CFIRET**: CFIRET is inserted after each call instruction to check that the function's

label is in the active set.

3. **CFIDEL**: CFIDEL is inserted before each return instruction to remove the function's label from the active set.

A state machine ensures that every function call and return is followed immediately by a CFIBR or CFIRET instruction. Figure 2.1 illustrates the abstract design of HAFIX. The result of this transformation is illustrated in pseudo-code below:

```
//main - label=0
int main()
{
    CFIBR 0 // insert label 0 into the active set
    ...
    foo();
    CFIRET 0 // ensure label 0 is in the active set
    ....
    CFIDEL 0 // delete label 0 from the active set
    return;
}

//foo - label=1
int foo()
{
    CFIBR 1 // insert label 1 into the active set
    ...
    CFIDEL 1 // delete label 1 from the active set
    return;
}
```

In the example above, foo can return to main since main's label, 0, is present in the active set when foo's return instruction is executed. However, main cannot return to foo, as foo's label, 1, is not present in the active set when main returns. Before returning all functions remove their label from the active set to ensure future returns cannot target the function.

Figure 2.2 details the state machine model of HAFIX.

HAFIX also includes two additional instructions for handling recursive function calls. CFIREC instructions replace CFIBR instructions at the beginning of recursive functions. CFIREC instructions increment a recursion depth counter stored in a hidden register called CFIREC_CTR and additionally activate labels when CFIREC_REC is set to 0. CFIDEL instructions decrement CFIREC_CTR if CFIFREC_CTR > 1 and remove the function's label from the active set when CFIREC_CTR equals 1. Figure 2.3 illustrates the abstract design of recursion handling in HAFIX. HAFIX does not support nested recursion. Our attacks (§ 5) do not use recursion.

We emphasize that our results apply only to Davi et al.'s x86 implementation of HAFIX, but not to their SPARC implementation [11]. Their x86 implementation uses the active-set policy for return instructions, while their SPARC implementation uses a full shadow stack

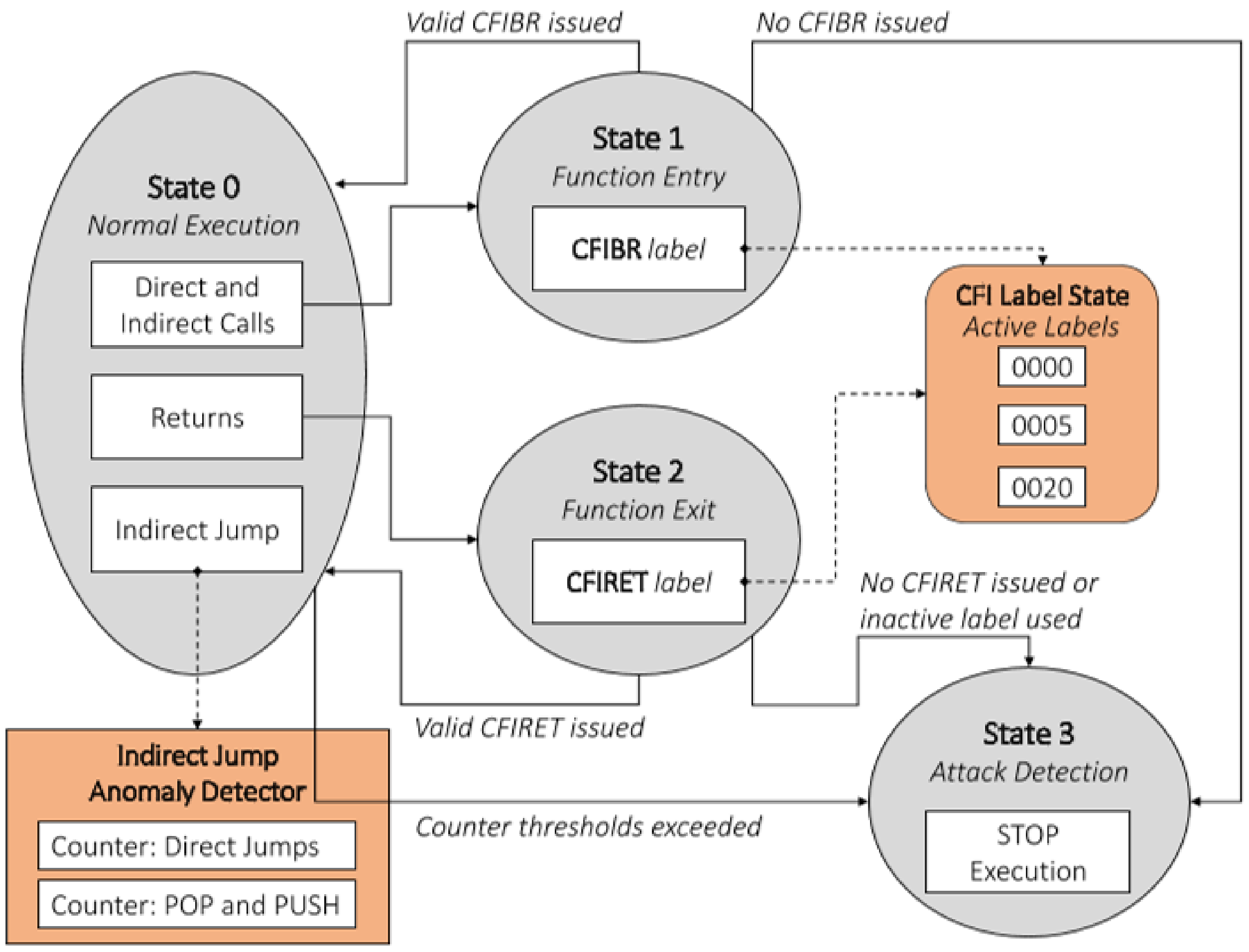


Figure 2.2: The HAFIX CFI State Model

	Forward-edge policy	
Backward-edge policy	Coarse-grained	Fine-grained
Coarse-grained	broken [2, 14, 12]	broken [3]
Active set	broken	(this thesis)
Shadow stack	broken [13, 3]	partially broken [3]

Table 2.1: Attacks against variations of CFI

for return instructions. Other researchers have studied shadow stacks; this thesis focuses solely on the active-set policy.

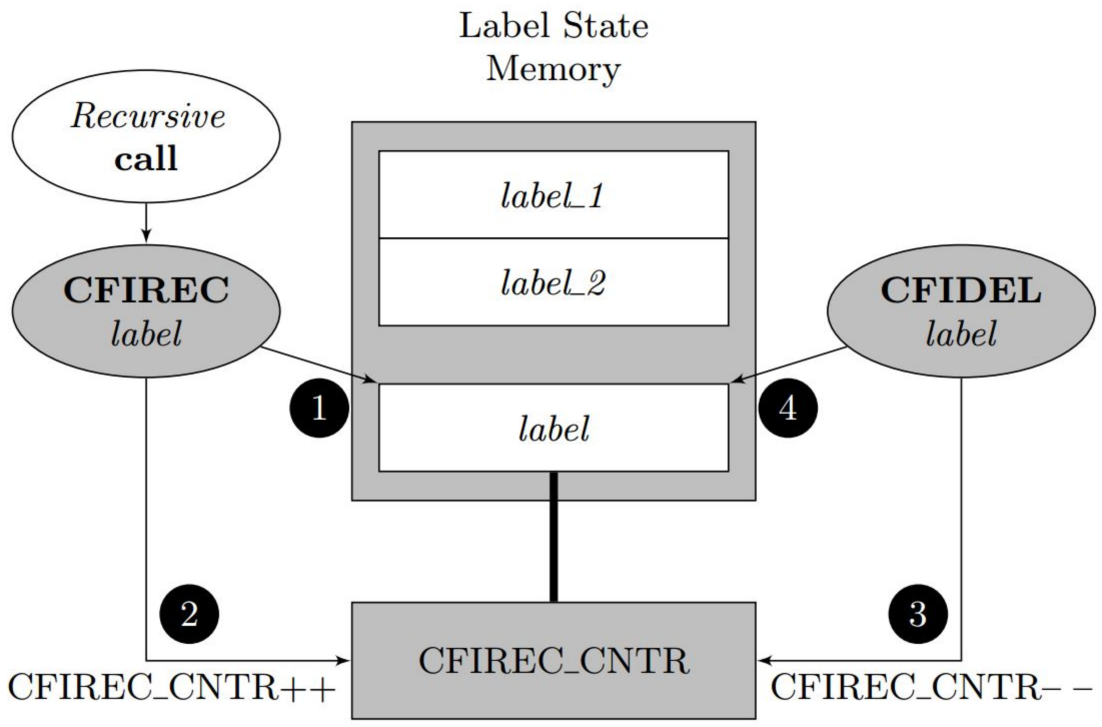


Figure 2.3: The Abstract design of recursion in HAFIX CFI

2.5 Attacks on CFI Flavors

Coarse-grained CFI includes a control flow graph with many additional execution paths beyond those intended by the programmer. Popular coarse-grained CFI implementations include ROPecker [4] and kBouncer [22]. Coarse-grained CFI has been bypassed and broken in previous research [2, 12].

Many researchers have studied fine-grained flavors of CFI [3, 13]. Carlini et al. found that fine-grained forward-edge CFI with a weak backward-edge policy (no shadow stack; allow returns to target any call-preceded instruction) can also be bypassed [3]. Even a shadow stack (the strongest possible policy for backward edges) can be vulnerable to code-reuse attacks in some cases [3]. Table 1 summarizes attacks on various CFI policies. Although HAFIX was not studied by Carlini et al., their findings imply that coarse-grained forward-edge CFI with an active set for backwards edges can be bypassed, as that policy is strictly weaker than coarse-grained forward-edge CFI with a shadow stack. Previous research has not evaluated the active-set policy for backwards edges with a fine-grained forward-edge policy. This thesis evaluates this CFI combination.

Chapter 3

Threat Model

3.1 Threat Model

Attacker Goals

Our adversary seeks to leverage a memory corruption vulnerability to execute arbitrary code. Through arbitrary code execution the attacker can exercise all permissions of a program and can execute system calls with attacker-supplied parameters.

Threat Model

Our main goal is to successfully execute a code-reuse attack on a system that implements Active-Set Backward-Edge CFI. In our model an attacker (1.) has full writable control of memory from a memory corruption vulnerability at least at one point during program execution, (2.) has full knowledge of the program's memory space including access to the program's code, and (3.) can bypass any code randomization by leveraging information leakage. We justify an attacker having full writable control of memory by verifying this is indeed true for the vulnerabilities in our case studies.

System Assumptions

We assume x86 HAFIX's active-set policy is deployed with the following additional defenses:

1. All indirect calls must follow the most restrictive static CFG for forward edges that still allows all feasible non-malicious execution [31]. (Thus, our attacks apply no matter what policy is applied to forward edges.)
2. Returns are restricted by the active-set policy: they can only target call-preceded instructions in active functions.

We also assume the following about our system:

1. DEP or an equivalent form of code-injection is enabled such that data is non-executable and code is non-writable. This is currently deployed on most systems by default.

2. ASLR or any code randomization is disabled or can be successfully bypassed by an adversary. Code randomization is an orthogonal defense to CFI, therefore evaluating ASLR would be irrelevant to our study of CFI.

Chapter 4

Active-Set CFI Attacks

The active-set backward-edge CFI policy contains weaknesses that can allow a malicious user to execute powerful attacks. The control flow graph of active-set backward-edge CFI contains many additional execution paths not present in the CFG of CFI with a shadow stack. More importantly, we found that many of the additional paths in active-set backward-edge CFI lead directly to powerful functions including exec. The design of active-set backward-edge CFI overlooked the frequency of powerful function calls occurring in functions likely to be on the call stack at any point in the execution of a program. Attackers can leverage these paths to exec to execute their attacks.

Five novel attacks on the active-set backward-edge CFI policy are presented in this chapter. The novel attacks are based on the following properties of the active-set policy:

1. The ability to return to any active function on the stack (not just the last function put on the call stack).
2. The ability to return to parent code in a child process after a fork.
3. The ability to return to earlier call sites in functions on the stack.
4. The ability to directly return to future call sites in functions on the stack.
5. The ability to return directly to the beginning of a program (typically the second call site in main).

4.1 Return-to-Active-Function Attack

The active-set policy allows return instructions to target any active function on the call stack. This property can be used by an attacker to directly return to any function in the call stack, bypassing any code residing in intermediate functions on the stack. These intermediate functions may contain code that is critical for secure execution. We show an example below.

```
int main(int argc, char *argv[]) {
    char *path;
    ...
    foo(path, argv);
    execl(path, argv); // (*)
    ...
}
int foo(char* path, char *argv[]) {
    ...
    vulnerable();
    if (validate(path) != 0) { exit(1); }
    ...
}
int vulnerable(char * argv[]) {
    char buf[1024];
    ...
    memcpy(buf, argv[1], strlen(argv[1]));
    ...
}
```

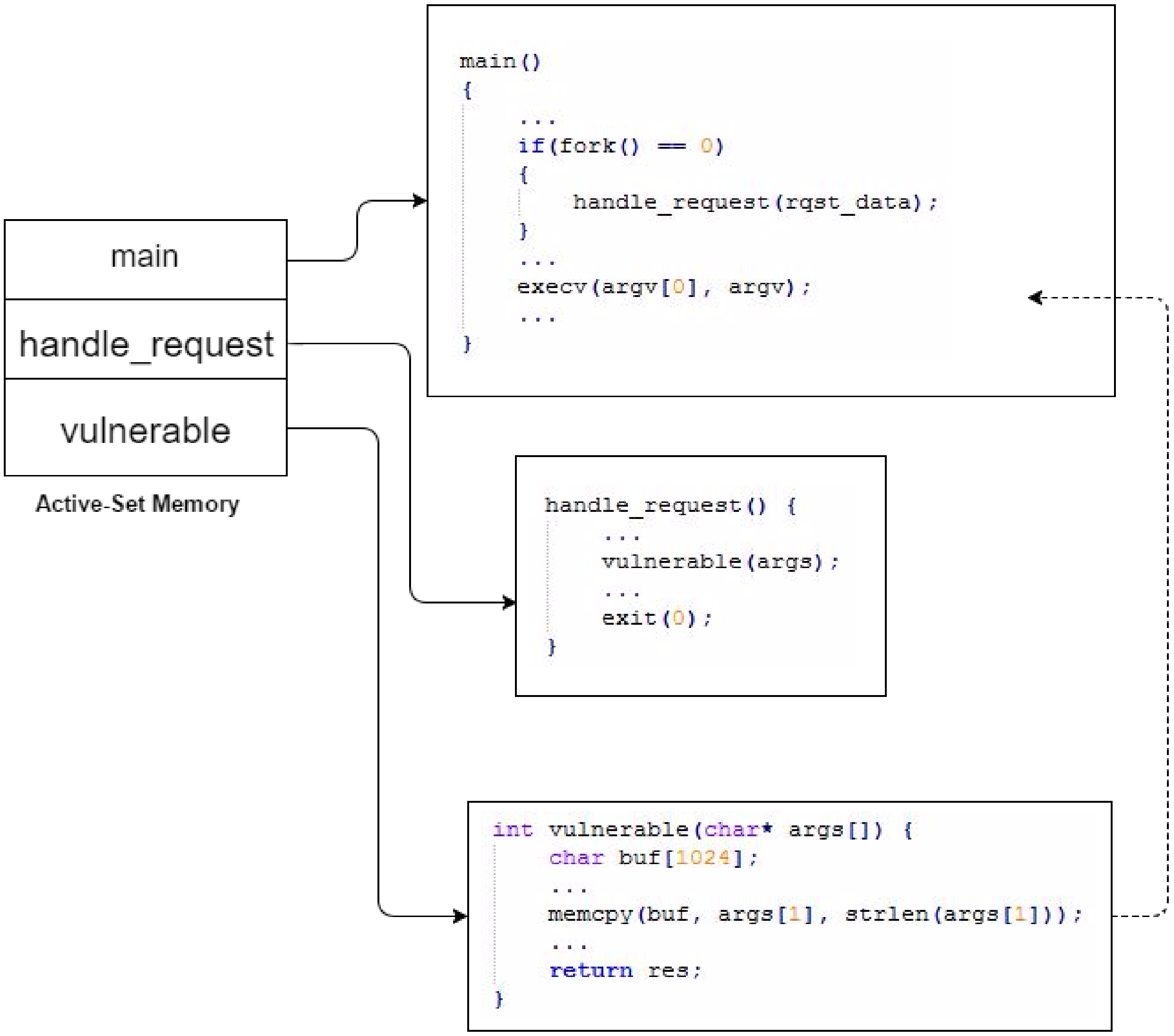
An attacker can leverage the vulnerability in vulnerable() to overwrite the return address to point to the statement marked (*) in main and overwrite the path variable to refer to a program of their choosing. By returning directly to main, the attacker bypasses the path variable validation that would have caused the program to exit.

4.2 Return-to-Parent-After-Fork Attack

Event loops and forked processes are common in server software. Servers often have a main process that waits for requests and forks a child process on each new request. In benign execution it is usually not possible to execute code that was designed for the parent process in the child process. The active-set policy allows an attacker who has compromised a child process to return to a function higher in the call stack (in the parent's region of the call stack) and execute code designed for the parent within the child process. This may enable a powerful attack, as often many unsafe library calls occur in code designed to be executed by the parent.

Figure 4.1 depicts a simple return-to-parent-after-fork attack. In the depicted attack, an exit call in the child process after the fork confines the child process to code reachable from the handle_request function. All code in main (with the exception of handle_request and exit) is only intended for the parent. The active-set policy enables a malicious user with control over memory in the child process to reach the execv call by overwriting the return address in vulnerable to directly return to the execv call in main.

Davi et al.'s x86 HAFIX implementation is intended for bare metal code and does not support multiple processes or fork. This attack is not applicable to that implementation, but it is applicable to any system that uses the active-set policy and supports multiple processes.



```
main()
{
    ...
    if(fork() == 0)
    {
        handle_request(rqst_data);
    }
    ...
    execv(argv[0], argv);
    ...
}
```

```
handle_request() {
    ...
    vulnerable(args);
    ...
    exit(0);
}
```

```
int vulnerable(char* args[]) {
    char buf[1024];
    ...
    memcpy(buf, args[1], strlen(args[1]));
    ...
    return res;
}
```

Figure 4.1: Simple Return-to-Parent-After-Fork Attack

4.3 Back-Call-Site Attack

A consequence of assigning unique labels to functions as opposed to individual call sites is that attackers who control a return address can return to call sites that appear earlier than the original call site if they are in the same function. This enables attackers to reach points in active functions that have already completed execution and are not intended to be re-executed. We show an example below.

```
int main() {
    char path[1024];
    ...
    strcpy(path, "/usr/bin/whoami");
    execl(path, arg);
    ...
    vulnerable();
}
int vulnerable() {
    char buf[1024];
    ...
    memcpy(buf, input, strlen(input));
}
```

The vulnerability in the vulnerable function can enable an attacker to execute execl in main with malicious arguments by overwriting the path variable and return address to target the execl in main.

4.4 Forward-Call-Site Attack

Similar to the Back-Call-Site Attack, an attacker can also return directly to call sites that occur later in an active function than the original call site, bypassing code occurring in between the original call site and the attacker's chosen call site. An example is shown below.

```
int main() {
    char path[1024];
    ...
    vulnerable();
    ...
    //Some code the attacker must avoid
    ...
    execl(path, arg);
}
int vulnerable() {
    char buf[1024];
    ...
    memcpy(buf, input, strlen(input));
}
```

The vulnerability in the vulnerable function can enable an attacker to execute execl in main with malicious arguments without needing to execute any code in between the vulnerable() call and the execl call. This attack is implemented by overwriting the path variable and return address to target the execl in main.

4.5 Return-to-Main Attack

The back-call-site attack can be combined with a return-to-active-function attack targeting the main function. Programs typically start and complete execution in the main function. As a result, main is marked active throughout the duration of the program, and all code (other than dead code) is reachable via some path starting in main. Suppose an attacker wants to reach code in function g, g is reachable via some path from function f, and main calls f. Then an attacker controlling any return address can always return to any call site in main that precedes the call to f and from there reach g.

Chapter 5

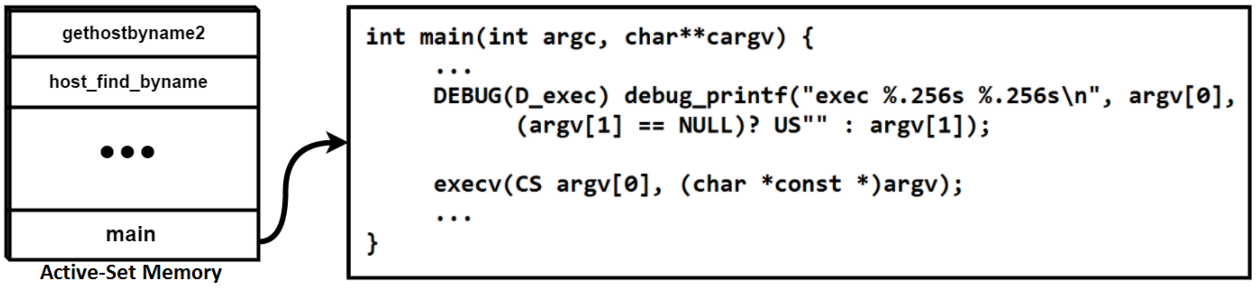
Case Study Evaluation

5.1 Motivation and Methodology

To understand the applicability of our attacks to real programs we select three programs with reported memory vulnerabilities and attempt to develop attacks on these programs under HAFIX. We select our programs by searching CVE databases for CVEs of open-source programs. We reproduce the vulnerabilities inside gdb to obtain an accurate backtrace and identify which functions are active at the point of the vulnerability. We also use gdb to write to memory and emulate an attacker's control over memory, and to verify that an attacker has full-writable control of memory for all programs.

5.2 Exim Mail Server

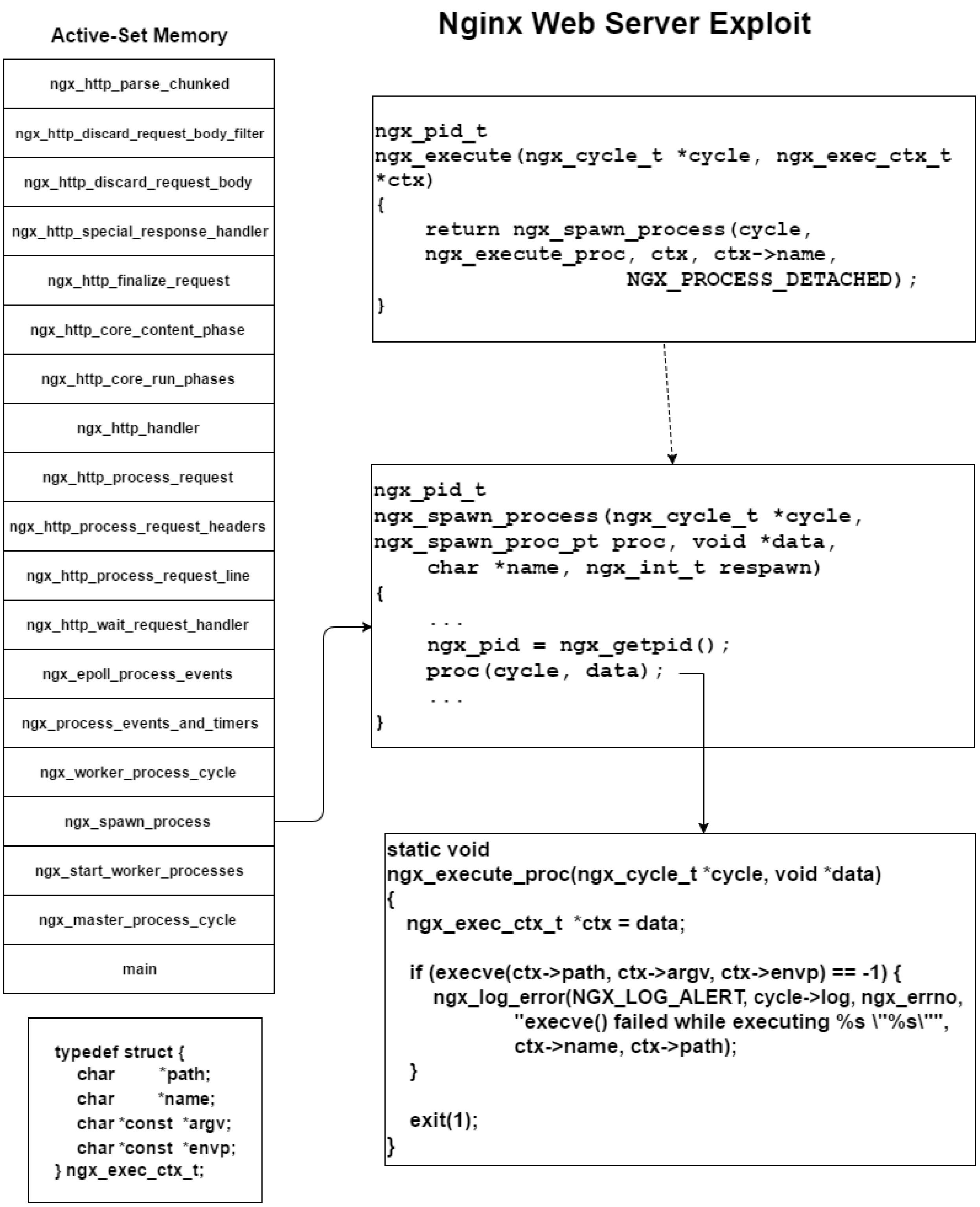
We examine a buffer overflow in the Exim mail server [26]. The vulnerability results from a heap based buffer overflow in the gethostbyname functions in glibc 2.2–2.18. We examine the vulnerability on a 64-bit Debian system.



```
int main(int argc, char**cargv) {
     ...
     DEBUG(D_exec) debug_printf("exec %.256s %.256s\n", argv[0],
           (argv[1] == NULL)? US"" : argv[1]);

     execv(CS argv[0], (char *const *)argv);
     ...
}
```

Figure 5.1: The active set for the Exim mail server during execution of the vulnerability with the execv call in main.


Active-Set Memory
Nginx Web Server Exploit
ngx_http_parse_chunked
ngx_http_discard_request_body_filter
ngx_http_discard_request_body
ngx_http_special_response_handler
ngx_http_finalize_request
ngx_http_core_content_phase
ngx_http_core_run_phases
ngx_http_handler
ngx_http_process_request
ngx_http_process_request_headers
ngx_http_process_request_line
ngx_http_wait_request_handler
ngx_epoll_process_events
ngx_process_events_and_timers
ngx_worker_process_cycle
ngx_spawn_process
ngx_start_worker_processes
ngx_master_process_cycle
main

```
typedef struct {
    char        *path;
    char        *name;
    char *const  *argv;
    char *const  *envp;
} ngx_exec_ctx_t;
```

```
ngx_pid_t
ngx_execute(ngx_cycle_t *cycle, ngx_exec_ctx_t
*ctx)
{
    return ngx_spawn_process(cycle,
    ngx_execute_proc, ctx, ctx->name,
                        NGX_PROCESS_DETACHED);
}
```

```
ngx_pid_t
ngx_spawn_process(ngx_cycle_t *cycle,
ngx_spawn_proc_pt proc, void *data,
    char *name, ngx_int_t respawn)
{
    ...
    ngx_pid = ngx_getpid();
    proc(cycle, data);
    ...
}
```

```
static void
ngx_execute_proc(ngx_cycle_t *cycle, void *data)
{
  ngx_exec_ctx_t  *ctx = data;

   if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
      ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                  "execve() failed while executing %s \"%s\"",
                  ctx->name, ctx->path);
   }

   exit(1);
}
```

Figure 5.2: The active set for Nginx is shown on the left. The ngx_spawn_process function contains a call to ngx_execute_proc which calls execve. ngx_execute_proc is called using the proc variable which can reference ngx_execute_proc when ngx_spawn_process is called from ngx_execute.

Control over Memory.

A security advisory [26] explains how an attacker can turn the gethostbyname buffer overflow into a write-anything-anywhere primitive. This satisfies our requirement that an attacker has full control of memory.

To summarize the advisory, they leverage the heap-overflow and partially overwrite the size field of the next contiguous free chunk of memory with a larger size, hence overlapping the free chunk with Exim's current_block mangaed by Exim's internal allocator. Exim's current_block is overwritten with partially arbitrary data, and control of both the pointer to the next block of memory in Exim's allocator and the data allocated work as our write-anything-anywhere primitive.

Exploitation.

We found that an attacker can bypass active-set CFI and perform an exec with an arbitrary command. We successfully spawned a shell while monitoring the program in gdb to ensure the active-set policy is respected. Our attack works by invoking an execv call in the main function. Because main is active when the gethostbyname vulnerability occurs, an attacker can use their control over memory to (1.) overwrite the return address to target the execv and (2.) overwrite the argument that is supplied to execv. Figure 5.1 shows the active functions at the point when the attacker controls memory and the call-preceded execv call in main.

0	gethostbyname2
1	host_find_byname
2	host_name_lookup
3	smtp_start_session
4	handle_smtp_call
5	daemon_go
6	main

Table 5.1: The full call stack of Exim Mail Server at the point when an attacker controls memory. The vulnerability occurs in the gethostbyname2 function.

Table 5.1 shows the full call stack at the point when an attacker controls memory.

Our exploit is an example of both the Back-Call-Site attack and the Return-to-Main attack described in the previous chapter.

5.3 Nginx Web Server

We study a integer overflow vulnerability in Nginx web server reported in CVE-2013-2028 [8]. We examine the Nginx 1.4.0 binary on a Debian 64-bit system.

Application	Attack techniques used	Exploitable with active set	Exploitable with shadow stack
Nginx Server	Return-to-parent, back-call-site	Yes	No
Exim mail server	Return-to-main, back-call-site	Yes	No
PHP	-	Yes	Undetermined (No write-what-where gadget found)

Table 5.2: A summary of our attacks. The second column indicates the attack methods we use in our exploits.

Control over Memory.

An integer signedness vulnerability in the decoding stage of Nginx allows an attacker to overflow an integer and trigger a stack-based buffer overflow. The overflow can be used to control arguments of a memcpy call, allowing an attacker to write arbitrary values to arbitrary locations [3]. Memory can be arranged after executing memcpy to return the process to accepting further requests without a crash. Carlini et. al study this vulnerability and report it is possible to write arbitrary values to arbitrary locations even when the program is protected with Shadow Stack CFI [3].

Exploitation.

We find an attacker can execute arbitrary code in the presence of active-set CFI. One of the functions in the active set when the memory vulnerability occurs, ngx_spawn_process, invokes a function pointer, proc, which can be overwritten by any value of the attacker's choice. An attacker with control over memory can (1.) overwrite the return address to target the proc function call in ngx_spawn_process, (2.) overwrite the proc function pointer to reference the ngx_execute_proc function, and (3.) overwrite the structure in memory used to hold the arguments for the execve call in ngx_execute_proc. Figure 5.2 summarizes our exploit and shows the active functions during the exploit. Overwriting the proc function pointer to reference the ngx_execute_proc function does not result in a forward-edge CFI exception as there exists another function, ngx_execute, that sets proc to ngx_execute_proc in non-malicious execution.

5.4 PHP

We investigate a stack buffer overflow in the sockets extension of PHP 5.3.6 that was reported in CVE-2011-1938 [7].

Control over Memory.

An attacker has full writable control of memory in the presence of active-set CFI. A memcpy call in the sockets function of php allows an attacker to trigger a stack overflow. The overflow can (1.) overwrite the arguments to a memcpy call in main and (2.) overwrite the return address to target the memcpy in main. The memcpy call in main is followed by an error condition check that returns when errors are detected. Memory can be overwritten to force a return through this error path to create a write-what-where gadget.

Exploitation.

We found that an attacker can execute arbitrary code despite active-set CFI. An attacker can leverage their control over memory to inject a php script of their choosing. The stack overflow occurs during execution of a php script, so the active set contains the required functions for execution of a php script. To execute an arbitrary php script, an attacker (1.) overwrites the existing php script in memory and (2.) overwrites the return address to target the php_execute_script function that executes php scripts.

5.5 Results

Table 5.2 summarizes our results. We believe the attacks we demonstrate are general and can be applied to other software.

Chapter 6

Enhancing Active-Set CFI

Our results demonstrate that Active-Set Backward-Edge CFI is broken and insecure. We propose two simple improvements to Active-Set CFI that eliminate the Return-to-Parent, Back-Call-Site, and Forward-Call-Site attacks. However even with these improvements, it is still unclear if the Return-to-Active-Function and Return-to-Main attacks alone are enough to break a CFI implementation. The only apparent mitigation to the Return-to-Active function and Return-to-Main attacks is using CFI with a shadow stack. Our results also add evidence to the claim that shadow-stack CFI is a necessary requirement for any secure CFI implementation. Table 6.1 summarizes our proposed improvements and the attacks they eliminate.

6.1 Adoption of Call-Site Labels

To prevent the back-call-site and forward-call-site attack, we propose assigning unique labels to individual call sites instead of functions. A compiler would then insert CFIBR instructions immediately before call sites instead of inserting CFIBR instructions at the beginning of functions. This modification restricts returns to target only the original call site in an active function. The attacks we demonstrate on Nginx and Exim servers are not possible under this modification.

6.2 Deactivation of Parent Function upon Fork

To prevent the return-to-parent-after-fork attack, we propose augmenting fork to clear the child process's active set before executing the child's code. The programs we evaluate do not contain programmer-intended paths in a child process that lead to functions made active in the parent process. We believe this holds true for most programs, however for compatibility we propose implementing this feature as an opt-out compiler option with a default of de-activating active parent functions. Our proposed compiler option can be modeled after the -fno-stack-protector option used in gcc to disable canaries.

Attack	Call-Site Labels	Parent Function Deactivation	Shadow Stack
Return-to-Active-Function	No	No	Yes
Return-to-Parent-After-Fork	No	Yes	Yes
Back-Call-Site	Yes	No	Yes
Forward-Call-Site	Yes	No	Yes
Return-to-Main	No	No	Yes

Table 6.1: A summary of proposed enhancements. The columns represent each proposed enhancement. Yes signifies that the enhancement prevents the attack.

6.3 Replacement of Active Functions with a Shadow Stack

We believe the only mitigation to the return-to-main and return-to-active-function attacks is the use of CFI with a shadow-stack. We were unable to find exploits that work in the presence of a shadow stack for the three programs in our case studies. Therefore, we believe the adoption of a LIFO shadow stack will be significantly stronger than an active set. Fortunately, Intel plans to add hardware support for shadow stacks in their upcoming Control-flow Enforcement Technology (CET) [23]. Recent research projects, including a successor to HAFIX, also present hardware support for shadow stacks [5, 30].

Chapter 7

Conclusion

Control-flow integrity implementations in the research literature have been proposed in both coarse-grained and fine-grained flavors. While coarse-grained CFI has been largely bypassed, bypassing fine-grained CFI implementations has remained questionable. Our work shows that the active set policy for backward edges can be defeated, no matter what forward-edge policy is used. These results suggest that the active set policy is too permissive and CFI needs to use a full shadow stack.

We hope our evaluation and attacks against HAFIX establishes a basis for stronger security policies in future CFI implementations. We also hope that the security benefits of implementing a full shadow-stack are further emphasized for researchers developing future proposals for CFI implementations.

Bibliography

[1] Martín Abadi et al. “Control-flow integrity principles, implementations, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), p. 4.

[2] Nicholas Carlini and David Wagner. “ROP is still dangerous: Breaking modern defenses”. In: *USENIX Security.* 2014, pp. 385–399.

[3] Nicholas Carlini et al. “Control-flow bending: On the effectiveness of control-flow integrity”. In: *USENIX Security.* 2015, pp. 161–176.

[4] Yueqiang Cheng et al. “ROPecker: A generic and practical approach for defending against ROP attack”. In: *NDSS 2014.*

[5] Nick Christoulakis et al. “HCFI: Hardware-enforced Control-Flow Integrity”. In: *ACM Conference on Data and Application Security and Privacy.* 2016, pp. 38–49.

[6] Crispan Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In: *USENIX Security.* 1998, pp. 63–78.

[7] *CVE-2011-1938.* Available from MITRE, CVE-ID CVE-2011-1938. May 2011. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1938`.

[8] *CVE-2013-2028.* Available from MITRE, CVE-ID CVE-2013-2028. May 2013. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028`.

[9] Thurston HY Dang, Petros Maniatis, and David Wagner. “The performance cost of shadow stacks and stack canaries”. In: *ACM Symposium on Information, Computer and Communications Security.* 2015, pp. 555–566.

[10] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. “Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation”. In: *Design Automation Conference.* ACM. 2014, pp. 1–6.

[11] Lucas Davi et al. “HAFIX: hardware-assisted flow integrity extension”. In: *Design Automation Conference.* ACM. 2015, p. 74.

[12] Lucas Davi et al. “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection”. In: *USENIX Security.* 2014, pp. 401–416.

[13] Isaac Evans et al. “Control jujutsu: On the weaknesses of fine-grained control flow integrity”. In: *ACM Conference on Computer and Communications Security*. 2015, pp. 901–913.

[14] Enes Göktas et al. “Out of control: Overcoming control-flow integrity”. In: *IEEE Symposium on Security and Privacy*. 2014, pp. 575–589.

[15] Trevor Jim et al. “Cyclone: A Safe Dialect of C.” In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[16] Arun Kanuparthi, Jeyavijayan Rajendran, and Ramesh Karri. “Controlling your control flow graph”. In: *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 43–48.

[17] Volodymyr Kuznetsov et al. “Code-pointer integrity”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 147–163.

[18] Bob Martin et al. “2011 CWE/SANS top 25 most dangerous software errors”. In: *Common Weakness Enumeration* 7515 (2011).

[19] Santosh Nagarakatte et al. “CETS: compiler enforced temporal safety for C”. In: *ACM Sigplan Notices*. Vol. 45. 8. ACM. 2010, pp. 31–40.

[20] Santosh Nagarakatte et al. “SoftBound: Highly compatible and complete spatial memory safety for C”. In: *ACM Sigplan Notices* 44.6 (2009), pp. 245–258.

[21] Aleph One. “Smashing the stack for fun and profit”. In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[22] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. “Transparent ROP exploit mitigation using indirect branch tracing”. In: *USENIX Security*. 2013, pp. 447–462.

[23] Baiju Patel. *Intel Releases New Technology Specifications to Protect Against ROP attacks*. Nov. 2016. URL: `https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks`.

[24] PAX-TEAM. “PaX ASLR (Adress Space Layout Randomization)”. In: (2003). URL: `http://pax.grsecurity.net/docs/aslr.txt`.

[25] PAX-TEAM. “PaX SEGMEXEC Documentation”. In: (2004). URL: `https://pax.grsecurity.net/docs/segmexec.txt`.

[26] *Qualys Security Advisory CVE-2015-0235*. URL: `https://www.qualys.com/2015/01/27/cve-2015-0235/ghost-cve-2015-0235.txt`.

[27] Ryan Roemer et al. “Return-oriented programming: Systems, languages, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), p. 2.

[28] Hovav Shacham et al. “On the effectiveness of address-space randomization”. In: *ACM conference on Computer and communications security*. 2004, pp. 298–307.

[29] Raoul Strackx et al. “Breaking the memory secrecy assumption”. In: *European Workshop on System Security*. ACM. 2009, pp. 1–8.

[30] Dean Sullivan et al. “Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity”. In: *Design Automation Conference*. ACM. 2016, p. 163.

[31] Caroline Tice et al. “Enforcing forward-edge control-flow integrity in GCC & LLVM”. In: *USENIX Security*. 2014, pp. 941–955.

[32] Minh Tran et al. “On the expressiveness of return-into-libc attacks”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 121–141.

[33] Chao Zhang et al. “Practical control flow integrity and randomization for binary executables”. In: *IEEE Symposium on Security and Privacy*. 2013, pp. 559–573.