École Polytechnique Fédérale de Lausanne

Evaluating Control-flow Hijacking Defences

by Benjamin Délèze

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Uros Tesic
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 14, 2020

Abstract

In the area of computer security, each new defence gives birth to a new attack, which in turn induces the creation of new mitigation. Control-flow integrity (CFI) was invented to try to counter attacks such as code reuse and return oriented programming. However existent CFIs are still incomplete and do not offer sufficient protection. The existing work is interesting but focuses too much on how many possibilities it blocks for a potential attacker and not enough on how many opportunities are left open. In this work, we build a benchmark that determines what exact protection is provided by a control-flow integrity mechanism. To do so, our benchmark will try to exploit flaws along with different control-flow transfers and see which ones are protected by the CFI. We tested our benchmark on the LLVM-CFI. The result is that even if some exploits are blocked, a lot of control-flow transfers are still completely vulnerable.

Chapter 1

Introduction

Computer security is a cycle. Attackers discover a new flaw and find a way to exploit it. Cybersecurity engineers close the gap...until a new flaw emerges. This is also true with more general methods. Code injection was countered by the invention and the deployment of data execution prevention. The attackers had to find new ways and they did with the idea of code reuse and return orientated programming. Control-flow integrity (CFI) was presented as a solution to these new methods. CFI analyzes the program to make a graph of its possible executions, it will then verify, during the execution, that the flow of the program stays in the graph.

There is a lot of CFI mechanisms available and it can be hard to choose one. Each of these CFIs have a speciality or a specificity that it emphasizes. In this work, we provide a benchmark that is easy to use and complete to help you choose a CFI. It is complete because our benchmark tests the CFIs on all the different control-flow transfers. We use the list of existing control-flow transfers given in Control-Flow Integrity: Precision, Security, and Performance [1]. To give more consistency to our benchmark, each control-flow transfer is tested with several types of attacks.

This work is interesting and needed because the existing work on CFI is about testing how many possible targets for attackers the CFI eliminate and the importance of the overhead produces by CFI. These works are also important but even if a CFI prevents the attacker from accessing 99% of the functions and gadgets, it is possible to exploit the remaining 1% to hijack the flow and this is what we will show in this work.

Chapter 2

Background

We will start by talking about stack defences (section 2.1), then we will continue with control-flow graph and control-flow integrity (section 2.2) and we will finish with the different types of control-flow transfer (section 2.3).

2.1 Presentation of the Stack Main Defences

We will quickly present the three stack defences that we tested in the benchmark. Before going further, it is important to understand the concept of a pointer. "Pointers are unstructured addresses to memory and a way to reference data or code" [5] Pointers are used in multiple ways. Some of the most important are to store the return address or the next instruction address. If the attacker gains control of one such pointer, it can redirect the flow of the program wherever it wants.

2.1.1 Data Execution Prevention

Before the deployment of Data Execution Prevention (DEP), there was no difference between code and data in most hardware. In this configuration, it was possible to inject malicious code at a memory location normally used for data. Then, if the attacker gains control of a function pointer, it can redirect the flow of the program to the code previously injected and execute it. DEP was invented to stop this type of attack. With some exceptions such as web browsers, the code is static. “DEP enforces code integrity, i.e., code cannot be modified or injected by an adversary.” [5] To do so DEP enforces a new policy which is: a memory page cannot be writable and executable. If the control-flow reaches a point that is marked as writable, like the data section, it will detect the attack. This mitigation works well to stop code-injection. However, DEP cannot protect against techniques which do not inject new code but reuse existing instructions (e.g. Return-Oriented Programming [7])

2.1.2 Stack Canary

The use of stack canary is the same as a canary in a mine. The Miners know that if the canary dies, there is a problem and they need to go out. This mitigation works on the same principle, it puts a secret value, the canary, before important pointers or after buffers. Before accessing the information that is after the canary, it will check that the canary remains unchanged. If it is not the case it will detect that a problem occurred. It is important to notice that this secret value is only initialized once at the beginning of the execution. Moreover, this mitigation only protects against continued overwrites, like buffer overflow. Arbitrary write will just bypass the canary [5].

2.1.3 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a mitigation that modifies the address space to place different blocks of memory at different places (e.g. the stack, the heap, the libraries) [5]. This mitigation makes the work of the attacker harder but does not counter it entirely. ASLR can be bypassed by brute force. Otherwise, if the attacker can leak an address, ASLR becomes useless.

2.2 Control-flow Graph and Integrity

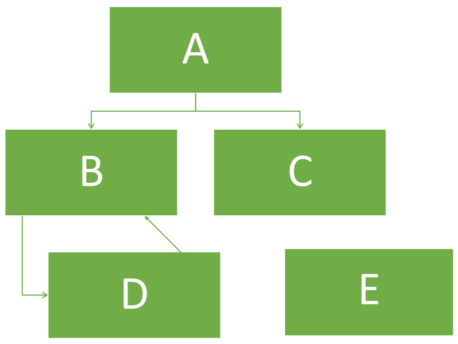


Figure 2.1: Example of CFG

The control-flow graph (CFG) is the graph that represents the possible paths that a program can take in its executions. Figure 2.1 shows that from block A it is possible that the flow goes to B or C. There is no way to block E, it is not reachable by the program. We can see that there are two types of edges: forward edges (A to B) and backward edges (D to B).

"Control-Flow Integrity (CFI) is a defense mechanism that protects applications against control-flow hijack attacks." [5] To achieve that, a CFI will first perform an analysis of the program. Then, during the execution, it will verify that the flow of the program respects the CFG. In figure 2.1, the flow is not supposed to reach block E. Likewise, the flow is not supposed to get from A to D without going through B. Unfortunately, the complexity of the analysis and the CFG of a program can quickly become huge and can lead to massive overheads. This is why some CFIs try to only approximate it whereas others use alternative techniques. They can determine that the set of valid targets for a given call is all functions or all functions with the same prototype.

We will work on the LLVM-CFI. This CFI especially enforces policies regarding indirect calls, casts, and virtual calls [9]. It has the advantages of having a very small overhead, less than 1%, and is included in the clang compiler [3].

2.3 Control-flow Transfers

We based our classification of the different control-flow transfers on the one realized in the paper Control-Flow Integrity: Precision, Security, and Performance [1].

They first separate the control-flow transfers into two kinds: backward control-flow transfer (return) and forward control-flow transfer when the flow is directed to a new location. However, this first classification is not sufficient and they divide the control-flow transfers into more categories.

1. Backward control-flow transfer
2. Forward control-flow using direct jumps
3. Forward control-flow using direct calls
4. Forward control-flow using indirect jumps
5. Forward control-flow using indirect calls supporting function pointers
6. Forward control-flow using indirect calls supporting Vtables
7. Forward control-flow using indirect calls supporting Smalltalk-style method dispatch
8. Complex control-flow to support exception handling
9. Control-flow supporting language features such as dynamic linking, separate compilation,etc.

Chapter 3

Design

3.1 Basic Exploitation Test

We use the principle of Basic Exploitation Test that is defined in Control-Flow Integrity: Precision, Security, and Performance [2]. This principle consists to use small programs that contain only what we need. Then we enable the CFI and try to see if it is still possible to exploit it. We will build one minimal program per test. If we find a way to exploit the minimal program, it shows that the CFI has a flaw regarding this type of attack and this control-flow transfer. But it is important to note that if the CFI blocks our exploits it is not proof that the CFI has no flaws.

3.2 Compatibility

The benchmark is built without hardcoded address for them to be compatible with other computers and CFIs.

However, some CFIs can instrumentalize the stack or change the memory layout to better enforce their policies. The benchmark is unfortunately not resistant to this kind of modification.

3.3 64 bits over 32 bits

Most modern computers have nowadays a 64 bits architecture. That is why we chose to develop our benchmark for a 64 bits architecture rather than a 32 bit one.

3.4 Stack Defences

To test the CFI, we compile our vulnerable programs without Data Execution Prevention (DEP) and Stack Canary and we do not activate Address Space Layout Randomization (ASLR). This was done to simplify our work. Letting all the defences, that are widely deployed and used, would have been possible. But it would have made our work more cumbersome. Moreover, it is not because they are widely deployed that they are always used. So, it is important to consider the

CFI as one independent tool and to evaluate it separately from the common defences. However, we have designed some tests to show how DEP, Stack Canary, and ASLR can be exploited.

In our design we test three different types of attacks:

1. Overwriting a neighbour variable
2. Hijacking the flow
3. Getting shell access by code injection

First, we have a code with two variables side-by-side, the attacker has the control of the first variable and the input is not correctly checked by the program such that we can overflow. The goal of this test is to see if by overflowing a variable we can modify the second despite the different mitigations of the stack.

The second test consists of modifying the target of a control-flow transfer to redirect the flow out of the control-flow graph. It is a similar test as the “inactive” test that we will use for judging the efficiency of a CFI.

The last test that we perform is to try to get shell access via code injection.

We are aware that we could have tests more types of attacks. In the beginning, we had the intention to also try Return Orientated Programming and Jump Orientated Programming. We also wanted to try some heap attacks. However, we finally preferred to focus on CFI testing. Moreover, we did not need these techniques to test the CFI because we made codes sufficiently vulnerable to be exploited by simpler methods.

We try to break different setups with these attacks:

1. Without mitigations
2. With DEP on, ASLR and Stack Canary off
3. With ASLR on, DEP and Stack Canary off
4. With ASLR on but with leaks, DEP and Stack Canary off
5. With Stack Canary on, DEP and ASLR off
6. With Stack Canary on but with a vulnerable fork, DEP and ASLR off

In the first setup, we disable DEP, ASLR, and Stack Canary to test that our exploits work well without mitigations. If an exploit does not work in this setup, discard all the following results in the different setups of this exploit.

Then, in situations 2, 3, and 5 we will activate only one mitigation at the time to test their effects on our exploits. The addresses and offsets are found via gdb, objdump, or trials and errors, but we do not use information leakage. The goal of these three setups is not that every exploit work but to see which of them fail when this or that mitigation is enabled.

In setup 4, we plan to leak some addresses. ASLR randomizes where some blocks of memory are placed. If we leak an address, we can deduce the other to exploit a vulnerable code. This setup is important for the rest of the work. We are going to use address leakage to automate the CFI’s tests to make them compatible on several computers.

Finally, in setup 6 we will use a vulnerability in the Stack Canary mitigation to find the canary and bypass it. The vulnerability is that the canary is initialized once when the program is launch. So, if we take the case of a server that does a fork on every new connection and crash on failure, we can guess the canary byte by byte. More explications will be given about how to guess the canary in the implementation section. We know that there are simpler methods to bypass the canary. We imposed ourselves this method because it interested us. However, implementing this method took us a lot longer than originally planned. That is why we only carried out this method for the hijacking-flow attack.

3.5 CFI Tests

We test if it is possible to hijack the flow of a program protected by a CFI. We consider that the CFI has failed to protect a certain type of control-flow transfer if someone manages to redirect the flow of a program protected by that CFI, using that type of control-flow at a location outside the control-flow graph. Most of the time we will test that by redirecting the flow to a function that is normally never called. However, to have more information on the precision of the CFI protection, we will try to redirect the flow to several different locations for each control-flow transfer. We will do several tests, but as long as on single one fails, the control-flow transfer is vulnerable.

Here are the different tests that we performed for each control-flow transfer. We did not necessarily use all of them for each control-flow transfer, you will find the exact details in the implementation part. We first test situations where the hijacked flow will be redirected to a "look-alike" function and then gradually move away from it with successive tests to see what margin is allowed by the CFI. For this purpose, the first test is almost no transgression of the control-flow graph, while the last test is to obtain a shell.

1. Other active function in the same branch but not in the good order
2. Other active functions with the same prototype
3. Other inactive function with the same prototype
4. Other inactive function with a different argument type
5. Other inactive function with a different return type
6. Shell

When explaining the different tests, we will generally talk about redirecting into different functions. However, we use this term for simplicity. In practice, we redirect the flow to a similar structure.

The first test we submit the CFI consists of a program where there are three code blocks as follows: the initial block A can either transfer the flow to block B or to block C. Moreover, there is a vulnerability in block A that allows us to hijack the flow and redirect it to B or C as we want.

The goal of this test is to launch an execution of the program where it normally does A and then B but we hijack the flow and force it to redirect from A to C.

The second test consists to redirect the control-flow to another function that has the same prototype that the function that should normally be reached by the flow after the transfer we hijacked. This function should not be directly reachable in the normal control-flow graph from the block of execution we are in but must be called once in another moment of the execution. That is why we call it “active”, in opposition to the next one.

The “inactive” test consists to redirect the control-flow to a function that has the same prototype that the function that is normally called by the flow-transfer that we hijacked. The particularity of this function is that it is never called normally. Note that it is sometimes necessary to write a call of the function in the code even if the call is never reached normally to avoid that compiler optimization removes it.

The two next tests are very similar to the “inactive” one. We use the same setup except that the prototype of the function we jump to is no longer the same. We first try to change the arguments of the function but keep the same return type. Then we do the inverse, so we keep the same arguments as the function that is normally reached by the flow of the program, but we modify the return type.

The last type of test that we perform is trying to gain shell access. We did it in several ways depending on the situation. We used code injection, but also vulnerable programs that already have a shellcode in their program or even return to libc.

Even so, we had ideas for more tests per control-flow transfer, we decided to stick only to those described above. The goal of our work was to evaluate the efficiency of a CFI and, as you will see in our results, our tests were enough to show that exploiting vulnerable programs is possible despite the tested CFI in all but one case.

We are aware that our tests are far from exhaustive, but we think they show a good panel of what is possible to exploit even if there is a CFI.

3.6 Leak and Vulnerable Code

For simplicity reasons, we decided to frequently leak addresses of buffers, functions, ... We took this decision to simplify our task and make the tests work on different computers. However leaking addresses do not impact the CFI evaluation because it does not modify the control-flow graph. Moreover, even if an address is not leaked there are still ways to obtain it through tools like gdb.

We allow ourselves to write very vulnerable code containing buffer overflows, dead code, hijackable memcpy. These different vulnerabilities are there to allow us to redirect the flow of the tested program. Indeed, the goal of our work is not to see if a CFI prevents buffer overflows and so on but if it detects when we redirect the flow outside the CFG.

3.7 Control Group

All the tests will be replicated twice, the first time, the binary will be compiled without the CFI being activated, and the second time with the CFI on. We do that to validate the effect of the CFI. If a test fails when we do not use the CFI, we need to discard the corresponding result when testing the CFI. Note that if a test succeeds without the CFI but fails when we activate it, it can be because the structure of the stack or of the heap is too modified for the test. In this case, further investigations are needed.

Chapter 4

Implementation

4.1 Stack Exploits

We describe shorty the structure of the exploits and the programs we made to test stack defenses. We also talk about the difficulties we encounter.

4.1.1 Overwriting Variable

Our first test is also the most basic. As all of our following tests, it consists of two files. First, the vulnerable program, in C, and then the exploit, in python.

The vulnerable program contains two variables side-by-side. The first, an array, is controlled by the attacker. By overflowing it, the attacker will try to modify the second variable.

We encounter one difficulty during the development of this exploit. Everything worked out as planned when we used GCC. However, when we try the same exploit with clang, the second variable was not overwritten anymore. We changed the order of the variables and even surrounded the target variable with two arrays.

In all the cases, the second variable was placed in memory such that it was just before the buffer(s). So it was not possible to overwrite it simply by overflowing the array. Clang reorder the variables to put the buffers after the simple variables.

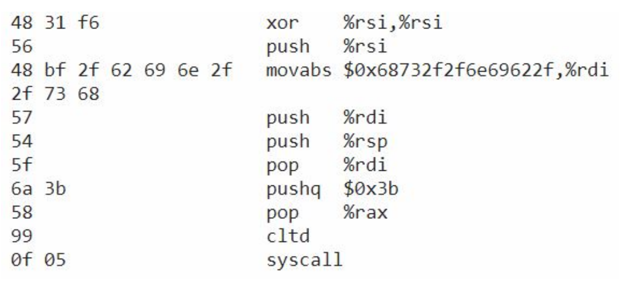
We finally overcome this problem by operating a slight change in the type of the target variable. We swap it to become an array of int containing a single element. By using this simple stratagem, clang was no longer reordering our variables and we could exploit our vulnerable program.

4.1.2 Code Injection

Even so, we knew that this type of attack was rendered obsolete by the use of DEP, we tested it to familiarize ourselves with shellcode. Besides, we think that a CFI manufacturer cannot be sure that DEP will be used along his CFI.

We decided to use gets because it was easier to inject shellcode than with scanf.

For the shellcode, we use a shellcode that can be found on packetstormsecurity.com



```
48 31 f6                xor    %rsi,%rsi
56                      push   %rsi
48 bf 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rdi
2f 73 68
57                      push   %rdi
54                      push   %rsp
5f                      pop    %rdi
6a 3b                   pushq  $0x3b
58                      pop    %rax
99                      cltd
0f 05                   syscall
```

Figure 4.1: The shellcode[6]

We also leak the address of the buffer, but we used it only for some tests. We received the address in big-endian format, so we have to convert it back to little-endian before adding it to our payload.

At this point, if our exploit works, we have a shell. It is easy to test that manually by entering some commands. However, it is a bit more difficult to test it automatically. The solution we chose consists of sending a touch command to the program after the point where the shell is normally pop. We can then check in our python program, with the "os.path.exists()" if the file was effectively created or not. If the answer is positive, then we can deduce that our exploits succeed. It is advisable to check before trying to create the file if a file with the same name already exists.

4.1.3 Flow Redirection

The goal of this attack is to redirect the flow of the program. To do so, we hijack a return such that it will transfer the flow to a function that is normally never reached.

We leak the address of the target function. We will use this information to bypass ASLR.

4.1.4 Stack Canary with Fork

To bypass stack canaries, we implemented a method involving the use of forks. At first, our intention was to implement this method for flow bending, code injection, and overwriting variable. However, it took us more time than expected, and, as stack exploit was just an introduction to our work, we decided to finally not finish all these exploits. In the end, we completed only the exploit for flow bending.

The exploit takes advantage of a vulnerability in the design of stack canary: the value of the canary is initialized only once, at the beginning of the execution of the program. This vulnerability is the most easily exploitable when a fork is present. The canary will remain the same in all threads. With this setup, we can find the value of the canary byte by byte. To do so, we overwrite only one byte of the canary until we find the right value for it. Then we can go to the next one.

Typically, on a 64 bits Linux, the canary is 64 bits long. So, if we try to brute force it at once we will have to test 2^{64} combinations. By guessing it byte by byte we only need 7*256 which is significantly lower. 7 and not 8 as we know that the first byte is 0x00 [10].

We used a simple UDP server with a fork on connection as a vulnerable program. Once we have the value of the canary, we can overwrite the return pointer and redirect the flow like in a flow redirection exploit.

In our first implementation, we used pwntools to run the vulnerable program, i.e. the server. However, the server powered by pwntools does not respond anymore after many requests even so it did not crashed. We now run the server in a separate terminal.

4.2 CFI Exploits

4.2.1 Return

The first type of control-flow transfer that we tested is the return. We implemented six different vulnerable programs and exploits according to the tests defined in section 3.5. All of the programs share the same structure :

- the hijacking-flow function where we will try to redirect the flow
- another function, that contains a vulnerable buffer overflow
- a third function, bar, where foo normally returns.

This buffer overflow allows an attack to overwrite the return address of the function. To simplify the automation of the tests, the vulnerable program also leaks the address of the hijacking-flow function.

4.2.2 Indirect Call

The method to exploit this control-flow transfer looks like the one for return (subsection 4.2.1). The difference is that instead of overwriting a return pointer, the exploit overwrites a function pointer. So a pointer that contains the address of a function and that is determined during the execution of the program.

4.2.3 Indirect Jump

This is almost the same as indirect call (subsection 4.2.2). Instead of overwriting a function pointer that is later "call", we overwrite one that is used in a jump.

To be sure to have the right type of jump, we use nasm [4]. This tool allowed us to write assembly directly in our code.

4.2.4 Direct Jump

Direct jump is a control-flow transfer that directs the flow to a specific location fixed in memory. To overwrite the address of the jump, we used a vulnerable memcpy.

The memory page where the code was situated was readable, executable but not writable. We had to change the permission of the page, with mprotect, to make the page writable. But the whole program was contained on a single memory page. So, we could not change the permission of this page as it was the active page. To solve this problem, we wrote some trash functions to fill the memory to have at least two pages. With that in place, we changed the permission of the page where the address that interested us was located.

Moreover, we used nasm [4] to get the right type of jump, with enough range.

4.2.5 Direct Call

We encounter the same obstacles as with direct jump (subsection 4.2.4). The solutions that we found are also the same. The only modification is that we use call and not jump so we had to adapt the structure of the program.

4.2.6 Longjmp/Setjmp

Setjmp and longjmp are two functions that work together. When calling setjmp, the function saves the environment in a variable of type jmp_buf . Then, when longjmp is called, it restores the environment saved by setjmp. These functions can be used to handle exceptions. The return instruction pointer (RIP) is saved in jmp_buf. The goal of this set of exploits is to change the saved value of the RIP such that, when longjmp is called, the flow is redirected elsewhere.

You cannot access the values contained in jmp_buf like you access the values of an array. Luckily it is still possible to access these values by converting the jmp_buf pointer to a void pointer and then incrementing it.

Another difficulty is that the data in jump_buf was encrypted. The RIP is XORed with a secret value and rotated left by 0x11 bits [8]. We found that the secret value was only initialized once. We could either try to guess the value via trials or errors with the help of a fork or chose to leak enough information to guess the secret value. For simplicity sake, we chose to leak values. We leaked the RIP of setjmp and the encoded value of the RIP. With these two values, we deduce the secret value and encode our own value correctly.

4.2.7 Global Offset Table

We change the address of the printf function in the global offset table (GOT) with one address of our choice.

It is possible to find the address of the GOT automatically with the help of this command: "readelf", "-S", "./filname".

It is important to populate the GOT with the address of printf before trying to overwrite this address.

4.2.8 Vtables

This control-flow transfer is the only one that we did in C++.

For this set of exploits, we overwrite the pointer to the Vtable of a variable with a pointer to another Vtable.

Chapter 5

Evaluation

5.1 Hardware and Software used

We developed our tests and programs on a Kali virtual machine. This was a Kali 2020.1 Rolling release with the 5.4.0-kali3-amd64 kernel version. We mostly used clang as a compiler, it was the version 8.0.1-4, but we also used GCC in its version 9.2.1 20200104 for stack exploits particularly.

Concerning the code, we wrote the code of the vulnerable programs in C except for the code regarding the Vtables control-flow transfer that is in C++.

We used python 2.7 and the pwntools library in order to write the different exploits. To inspects the memory and the binary, we used GDB with the GEF extension.

To ensure that our code works also on another computer, we tested it on a Ubuntu 18.04.04 LTS VM and we obtained the same results

5.2 Compile Lines

To tests our exploits in a different setup, we compiled them differently, depending on which stack defences we wanted to enable or not. You will find these different lines thereafter:

- Neither DEP nor Stack canary :

  ```
  gcc (or clang) -fno-stack-protector -z execstack -o foo foo.c
  ```

- DEP, but not Stack canary:

  ```
  gcc (or clang) -fno-stack-protector -o foo foo.c
  ```

- Stack canary but not DEP:

  ```
  gcc (or clang) -fstack-protector -z execstack -o foo foo.c
  ```

To disable ASLR we used:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

5.3 How to Recognize a Successful Exploit

In order to harmonize the benchmark, we quickly took care of building our vulnerable programs and our tests on models in order for them to be easier to understand. You can test each test individually, but the code is rather designed to perform all the tests related to the same control-flow transfer at once, or even all the tests at once.

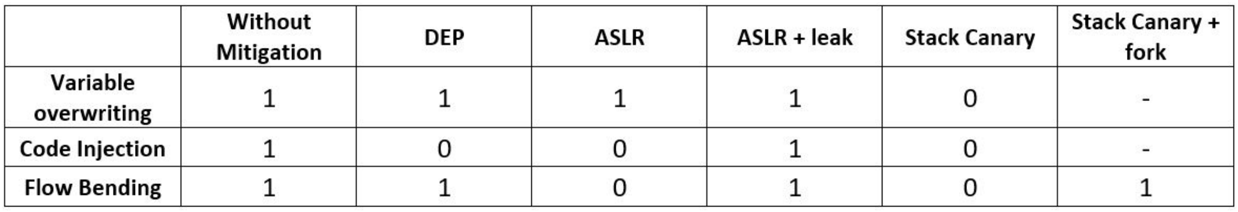
We stuck to two ways of telling the exploits that they are successful.

- The first one concerns the exploits where we try to redirect the control-flow to another function. In this case, a simple print is enough
- The second case comes when we try to get shell access. Our exploit send a “touch” command after having tried to pop a shell. Once it did that, it looks if a file with the corresponding name has been created. If it is the case, it removes the file and returns success. Note that we also check if the file exists before trying to exploit the program. In this case, we remove it and then try to get a shell normally.

5.4 Stack Evaluation

We evaluate the efficiency of the code injection, flow redirection, and overwriting variable with different setups:

1. Without mitigations
2. With DEP on, ASLR and Stack Canary off
3. With ASLR on, DEP and Stack Canary off
4. With ASLR on but with leaks, DEP and Stack Canary off
5. With Stack Canary on, DEP and ASLR off
6. With Stack Canary on but with a vulnerable fork, DEP and ASLR off



	Without Mitigation	DEP	ASLR	ASLR + leak	Stack Canary	Stack Canary + fork
Variable overwriting	1	1	1	1	0	-
Code Injection	1	0	0	1	0	-
Flow Bending	1	1	0	1	0	1

Table 5.1: Result of our tests on stack defences: 1 = exploit successful, 0 = exploit failure

5.4.1 Without Mitigation

The goal of this setup was to have a baseline on which to base our comparisons. So, it is a good thing that each of the three attacks is a success. If you reproduce our tests but, in your trials, one of the attacks fails, it is better to correct it before going anywhere further.

5.4.2 With DEP on, ASLR and Stack Canary off

With this configuration, code injection does not work anymore. This is entirely normal because we try to execute some code that we previously put in the data section and DEP handle precisely that. However, this has no effect on our two other attacks, that work still perfectly well.

5.4.3 With ASLR on, DEP and Stack Canary off

In this setup, the overwriting variable is the only one that works. ASLR randomizes the place of blocs of memory (e.g. the stack, heap, libraries). So, two adjacent variables of the stack will remain adjacent even if ASLR is enabled. Therefore, the overwriting variable exploit is still working.

However, the two other attacks are not working anymore. This is because until now, we got the addresses manually. Without ASLR these addresses remain the same between executions. This is not the case with ASLR, so our exploits try to make the program return to a wrong place and it makes it crash.

5.4.4 With ASLR on but with Leaks, DEP and Stack Canary off

The vulnerable program now leaks addresses. With this modification, all the attacks succeed. It is because, even with ASLR on, our exploits know where to make the program jump to exploit it.

Note that this modification has no impact on the overwriting variable exploit.

5.4.5 With Stack Canary on, DEP and ASLR off

With this configuration, all our attacks fail. This is because stack canary detects when a buffer overflow tries to overwrite the following variable. As we do nothing do handle the canary, it is normal that our attacks are detected and denied.

5.4.6 With Stack Canary on but with a Vulnerable Fork, DEP and ASLR off

As specified on the implementation part (subsection 4.1.4), we only build an exploit like this for the flow bending attack. This exploit proves itself successful as it guessed the canary. By doing so, we can redirect the flow of the program to a function of our choice.

5.4.7 Conclusion on Stack

As trying to bypass stack defences was just an introduction for us, we did not try to enable several defences at a time. However, we can deduce that:

- if DEP is enabled, it will block code injection but have no impact on other types of attacks.
- ASLR is defeated by leaking an address, the combination of other defences will not change that.

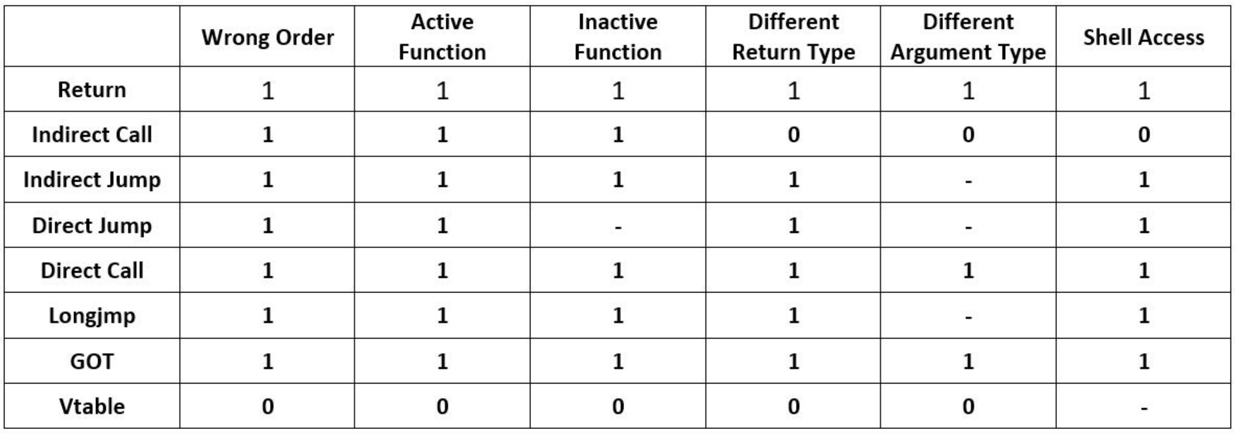
- Stack Canary can be defeated via the fork method or by leaking the canary, combining it with ASLR will not radically change that. It will just make it a bit more cumbersome for the attacker.

5.5 CFI Evaluation

We tested our results several times in order to be sure that they were consistent, and it was the case. We also tested our exploits both on Kali and Ubuntu. The results that we obtained were the same regardless of the operating system.

All the tests are realized twice, one without a CFI, one with a CFI. In our case, we chose to evaluate the CFI that is included in LLVM. We recall that we operate without DEP, ASLR and any Stack Canary.

From a personal point of view, we were surprise by the number of exploits that work even if the CFI enabled.



	Wrong Order	Active Function	Inactive Function	Different Return Type	Different Argument Type	Shell Access
Return	1	1	1	1	1	1
Indirect Call	1	1	1	0	0	0
Indirect Jump	1	1	1	1	-	1
Direct Jump	1	1	-	1	-	1
Direct Call	1	1	1	1	1	1
Longjmp	1	1	1	1	-	1
GOT	1	1	1	1	1	1
Vtable	0	0	0	0	0	-

Table 5.2: Result of our tests on CFI: 1 = exploit successful, 0 = exploit failure

5.5.1 Return

We performed the standard tests for this control-flow transfer. We saw in the evaluation of the LLVM-CFI made by [1] that LLVM-CFI has no support for backward-edges. Knowing that, our results are not surprising.

5.5.2 Indirect Call

Half of our exploits are blocked by the CFI. We decided to not conduct further tests because our goal is to demonstrate that there are flaws in the LLVM-CFI. As three of our attacks succeed, it is enough.

We can make the hypothesis, by looking at which attacks succeed and fail, that the LLVM-CFI, for this particular type of control-flow transfer, considers as a valid target every function that have a given prototype.

5.5.3 Indirect Jump

As we do not always jump to function with this type of control-flow transfer, here as the tests that we performed:

1. The program normally jumps to a label and from that point prints three sentences, we try to exploit the flow, so it jumps directly to the label corresponding to the third sentence.
2. The program contains a label that can be reached in some executions but not in the one that we test. The goal is to hijack the flow in order to reach this label.
3. The program contains a label that is never reached. By hijacking the flow, we want to reach it.
4. The program has a function, hijacking-flow, that is never called. We will try to redirect the flow from a jump to this function.
5. We redirect the flow of the program from the jump to the input buffer that will contain a shellcode.

The LLVM-CFI blocks none of our exploits.

5.5.4 Direct Jump

Here are the tests that we performed:

1. We hijack a jump to an "if" in order to jump to another condition of the same "if/else" structure.
2. We hijack a jump to an "if" to jump to another point, outside of these "if/else", but in the same function.
3. We hijack a jump to an "if" to jump to another function.
4. We hijack a jump to a fixed label in order to jump to a shellcode. In order to be reachable, we used malloc to insert the shellcode in the heap.

The LLVM-CFI blocks none of our exploits.

5.5.5 Direct Call

We performed the standard tests for this control-flow transfer.
The LLVM-CFI blocks none of our exploits.

5.5.6 Longjmp/Setjmp

We implemented five different tests to see if the CFI prevents us to exploit setjmp/longjmp :

1. We replace the encrypted RIP by the RIP of another longjmp
2. We redirect the flow to a label in the same function (main)
3. We redirect the flow to a function that is in the control-flow graph
4. We replace the encrypted RIP by the address of a function that is never called
5. We get a shell by redirecting the flow to a heap buffer containing a shellcode.

The LLVM-CFI blocks none of our exploits.

5.5.7 Global Offset Table

As this type of attacks is well known and it even exists mitigations like RELRO, we thought that the CFI may detect them. It turns out that this is not the case.

The LLVM-CFI blocks none of our exploits.

5.5.8 Vtables

These tests were the last that we did, but also the only ones that really turn out to show the efficiency of the CFI.

We saw in the documentation of the CFi [9] that the LLVM-CFI is focused on Vtables, so the result is not too surprising, but we wonder why it detects better these types of attacks than the ones with indirect call, where it is also supposed to do well. Unluckily we had no time to investigate further, but it may be interesting to see if the CFI is more performant in that case because it is C++ or for another reason. It would be interesting to perform further tests on Vtables in order to see if we could manage to bypass the CFI.

Chapter 6

Related Work

Control-Flow Bending: On the Effectiveness of Control-Flow Integrity [2]: By using a basic exploitation test, the authors show that CFI still has vulnerabilities and that it is possible to exploit them. It concludes that the use of a shadow stack is mandatory to achieve good security and that CFI gets some results but that it is not perfect.

Control-Flow Integrity: Precision, Security, and Performance [1]. In these paper, the authors compare a lot of CFI using different axis like performance, supported control-flow, the precision of the analysis concerning forward and backward edges. With their classifications, they have gone beyond the simple classification of CFI between "coarse-grained" and "fined-grained".

While the two papers test specific CFIs for vulnerabilities, our work is meant to be a general benchmark to be used to test both old and new CFI mechanisms.

Chapter 7

Further Work

Several things can be made to improve this project:

- Measure the overhead of the tested CFI.
- Add tests for Smalltalk-style direct call.
- Improve the tests to support CFIs that modify the structure of the memory.
- Test this benchmark on more CFI. This was made complicated due to the lack of updates and of documentation on research's CFI that we found.

Chapter 8

Conclusion

We have built exploits for eight different control-flow transfers. For each of this control-flow transfers we gradually moved away from the CFG. On top of that, we also made some tests on stack defences to see how it was possible to bypass DEP, ASLR, and stack canary.

Our exploits on the stack showed us that even if some type of attacks were countered by the mitigations, like code injection by DEP, it was still possible to bypass them using other types of attacks or by having programs with vulnerabilities such as address leaks. Once we were sure of that, we serenely disabled those defences to focus on CFI.

We tested our exploits on the LLVM-CFI. The results of our tests show that LLVM-CFI efficiently protect against attacks on Vtables and only partially against attacks on indirect calls. We still hijacked the flow using indirect calls as long as we redirected it into a function that has the same prototype as the right one. But the LLVM-CFI is completely inefficient with respect to the other control-flow transfers.

The results show that the developed benchmark can quickly evaluate the protection offered by a CFI according to several attack types along with different control-flow transfers.

Bibliography

[1] Nathan Burow, Scott Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Computing Surveys* 50 (Feb. 2016). DOI: `10.1145/3054924`.

[2] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.R. Gross. “Control-flow bending: On the effectiveness of control-flow integrity”. In: *24Th USENIX Security Symposium (USENIX Security 15)* (Jan. 2015), pp. 161–176.

[3] *Clang: a C language family frontend for LLVM.* URL: `https://clang.llvm.org/`. (accessed: 04.06.2020).

[4] “Installing and Using NASM”. In: *Guide to Assembly Language Programming in Linux.* Boston, MA: Springer US, 2005, pp. 153–166. ISBN: 978-0-387-26171-3. DOI: `10.1007/0-387-26171-0_7`. URL: `https://doi.org/10.1007/0-387-26171-0_7`.

[5] Mathias Payer. *Software Security. Principles, Policies, and Protection.* 2019.

[6] Rajvardhan. *Linux/x64 execve(/bin/sh) Shellcode.* URL: `https://packetstormsecurity.com/files/153038/Linux-x64-execve-bin-sh-Shellcode.html`. (accessed: 20.03.2020).

[7] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”. In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012). ISSN: 1094-9224. DOI: `10.1145/2133375.2133377`. URL: `https://doi.org/10.1145/2133375.2133377`.

[8] *sysdep.h.* URL: `https://github.molgen.mpg.de/git-mirror/glibc/blob/20003c49884422da7ffbc459cdeee768a6fee07b/sysdeps/unix/sysv/linux/x86_64/sysdep.h`. (accessed: 04.06.2020).

[9] C. Tice, T. Roeder, P. Collingbourne, Stephen Checkoway, Ú Erlingsson, L. Lozano, and G. Pike. “Enforcing forward-edge control-flow integrity in GCC LLVM”. In: *23rd USENIX Security Symposium (USENIX Security 14)* (Jan. 2014), pp. 941–955.

[10] Adam ’pi3’ Zabrocki. *Scraps of notes on remote stack overflow exploitation.* URL: `http://www.phrack.org/archives/issues/67/13.txt`. (accessed: 27.03.2020).