Control Flow Integrity using AES encryption and Merkle trees

Christopher Gauffin
Joseph Johansson
Jesper Lagnelöv
Kevin Harrison

January 18, 2023

Contents

1 Introduction

It is still a common problem today that programs are vulnerable to control flow attacks, where an attacker redirects the flow of execution which could effectively give them unauthorized access, control of an entire system, or access to sensitive data. Most operating systems have protections by default against this since it is a widely known problem. Yet a smart attacker may know how to circumvent the guards put in place as well as exploit code written by an inexperienced programmer. Most protection mechanisms also focus on patching attack techniques rather than on the fundamental problem itself. We assume that there is a stack overflow and prevent it by implementing stack canaries, we use non-executable code to prevent code injections and address space layout randomization (ASLR) to address information leaks. This is why we consider Cryptographically Enforced Control Flow Integrity (CCFI) [1] to be the state of the art to today, where complete control flow integrity is achieved by using cryptography, addressing the fundamental problem instead. This paper was used as a baseline for our project, but where we chose a slightly different approach because of a potential flaw concerning replay attacks, explained in subsection 3.2. This method was implemented within the minix kernel using a daemon server combined with a user-space library explained in section 4. There are many ways of breaking control flow, we focus on buffer overflow attacks, more specifically overwriting variable pointers and function return addresses that are pushed onto the stack.

2 Background

2.1 Buffer overflow attack

The memory typically has a layout consisting of a kernel space, where the minix kernel runs. This is followed by the stack growing downwards which holds the local variables which are pushed onto the stack for each function call, and a heap growing upwards allocating larger more persistent objects. The lowest addresses are for the data section, global variables, and a text section, the actual compiled machine instructions of the program to be executed, see Figure 1.

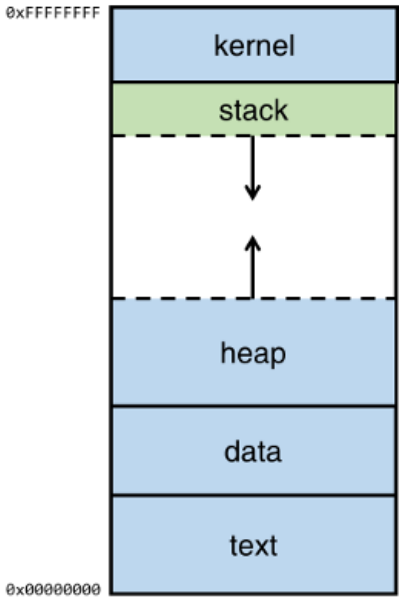

Figure 1: Memory layout

A stack frame belongs to a function and consists of its parameters, return address, base pointer (ebp), followed by local data such as a buffer or variable, see A) in Figure 2. However, when variables or addresses already present on the stack are overwritten control flow can easily be broken. This can happen when a buffer for example is allocated to be 100 bytes, but then 105 bytes are written to it, which is going to

overwrite 5 bytes of data that is written to addresses above the buffer, see B) in Figure 2. If the overflow is aligned correctly, then the entire base pointer and return pointer can be overwritten. This makes it possible for an attacker to redirect the flow by either making the function return to some other more sensitive function or to some maliciously injected code. Alternatively, some local pointer could be overwritten, so that when it is dereferenced, some entirely different memory address might be written or read to instead.

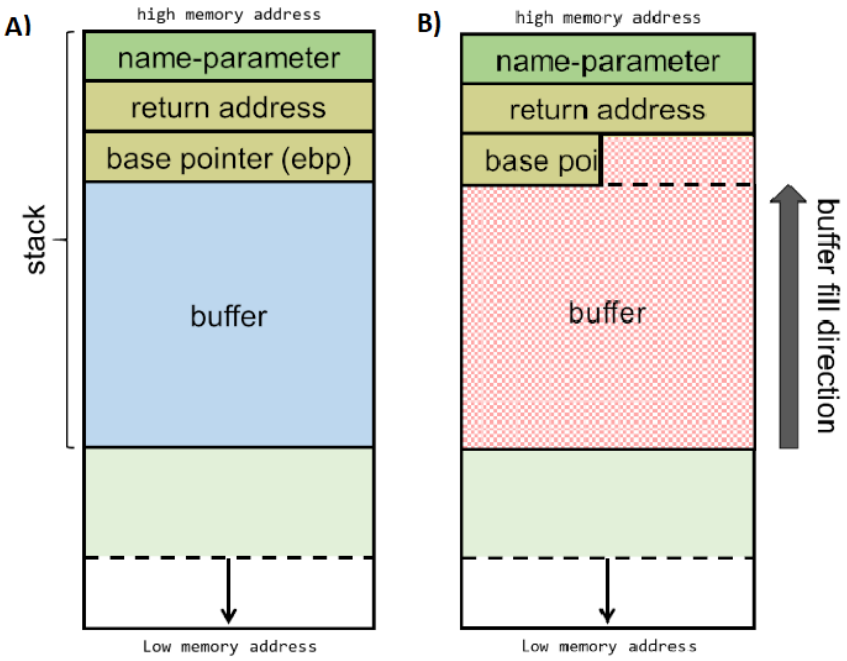

Figure 2: A stackframe before and after buffer is overflowed

2.2 Stack canaries

A common technique to prevent return addresses from being overwritten is to add a stack canary to the stack frame just before the base pointer. This canary is 4 bytes of random data, which is not known to the attacker. Figure 3 The value of the canary will be checked just before the return of the function. If the return address would have been overwritten, then the canary must also have been overwritten and the overflow is detected when the check fails. However, if the attacker knows the canary, they could overwrite the correct 4 bytes and change the return address undetected. Brute-force would be too time-consuming, as it would require 1 billion guesses. But if each byte is tested iteratively you would only require 255 * 4 = 1024 tries in the worst case. This is accomplished by only overwriting the byte that is being tested, if it fails it is the wrong number, but if the program does not crash we know that that byte is correct and can move on to the next. So stack canaries make it harder for an attacker but are not sufficient protection.

Figure 3: A stack canary

2.3 ASLR

Address Space Layout Randomization (ASLR) is used to randomize the memory layout, re-arranging the sections, stack, heap and libraries to random positions. This makes it much more inconvenient for an attacker to do a buffer overflow, since they rely on exact positions in memory of functions and variables. If

a routine of a function would be moved on every execution then the attacker would need to somehow leak that information, and do a lot more memory analysis in order to find the address to which they want to overwrite a return address. It is a good protection mechanism, but ASLR-protected addresses can in fact be leaked. This could be accomplished by abusing the branch predictor buffer of the CPU or the memory management unit or by looking at addresses of libraries to find the ASLR offset.

3 Design choices

3.1 State of the art

As previously mentioned, this project builds CCFI as explained in a paper by Mashtizadeh et al.[1], which is considered state of the art. Mashtizadeh et al. work with the assumption that an attacker has read/write access to the entire memory. The idea behind CCFI is to append a message authentication code (MAC) to every jump address written to memory. The MAC is calculated in the following way:

$$\text{MAC}(K, \text{pointer, class})$$

Where k=key(in CPU register), pointer=pointer to protect, class=pointer's address

Before trusting an address initiating a jump, the MAC address is verified and if it is correct then the jump will be made, otherwise the program is crashed since tampering has been discovered. This should prevent an attacker from tampering with the addresses written in memory with the goal of making the program jump to arbitrary memory address. The paper suggests that AES should be used since many CPU:s feature special hardware for fast AES calculations, reducing some of the overhead (estimated to be 3-18%) introduced when calculating a MAC for every jump address.

3.2 Limitation of state of the art

While the CCFI solution prevents an attacker from inserting arbitrary (yet valid) pointers and return addresses using MACs, it is vulnerable to replay attacks. This is because the pointer's *class* is acting as a nonce in the MAC despite the potential for the *class* to be reused.

For example, this scenario arises if a stack frame's return address a is added to the stack alongside its MAC $\text{MAC}(K, a, \&a)$, the program continues to execute and at some point creates a new stack frame with return address b and MAC $\text{MAC}(K, b, \&b)$ where the memory location of the return address is the same as the previously mentioned stack frame's ($\&a = \&b$). In this scenario an attacker can overwrite b and $\text{MAC}(K, b, \&b)$ with a and $\text{MAC}(K, a, \&a)$. The CCFI system will then validate pointer a potentially breaking control flow integrity.

Pointers stored in text and data sections of the program don't have this issue as their memory locations stay the same throughout the execution of the program. This is not true for the stack and heap.

The CCFI paper's addresses this shortcoming by adding a random offset to each stack frame and heap allocation. This is different to ASLR in that the offsets added at each frame/allocation, not just during the initialization of the stack and heap. This makes it less likely for pointer addresses to repeat, but gives no guarantees.

3.3 Mitigating replay attacks

Replay attacks can be eliminated by introducing a true nonce. For stack frames this is possible by assigning a unique id to each frame (for example an increasing counter). If the current stack frame is kept track of at runtime, any MACs used in a frame would then be invalid to use in other frames. This solution does nothing to address the heap and in the context of a single stack frames memory addresses still must not be reused. Furthermore, which id's are valid cannot be stored naively in program memory as an attacker

may manipulate it. They must be stored in protected hardware (as the MAC key is in the CCFI solution) which due to the large number of ids would be infeasible.

Our solution addresses both of these problems by keeping track of individual pointers in a constant amount of protected memory using a Merkle tree (see subsection 4.1).

4 Implementation

Our solution protects the memory pointers of a program during its execution despite the presence of an attacker with read access to readable program memory and write access to writable program memory. This is achieved by maintaining a Merkle tree to protect pointers using a constant amount of protected memory. The protected memory is modelled as a MINIX server called the Control Flow Manager server (CFM server) which communicates with the program during execution. The maintaining of the Merkle tree is done by injecting code into the program at compile-time which we roughly model with C macros.

4.1 Merkle tree pointer protection

A Merkle tree is a tree data structure where a hash chain is formed by each interior node in the tree containing a hash of its children [2]. Merkle trees allow a trusted source to express an authenticated commitment to a set of elements with a constant amount of memory, namely with the root of the Merkle tree. In our implementation the trusted source is the CFM server, but ideally the trusted source would be a protected CPU register similar to the CCFI paper. Furthermore, inclusion and exclusion queries can be calculated in $O(\log(n))$ time.

We use these properties to solve the problem of keeping track of which pointers on the stack and heap are valid at a specific time in the execution of a program. The program maintains a Merkle tree of the current protected pointers in its own memory. In the following, steps 1-2 describe querying and 1-4 describe updating the tree:

1. Query the CFM server for merkle root and check it matches tree in memory's root.
2. Binary search (with &pointer as the key) through the tree for location of node with payload (pointer, &pointer). Validate that the search path hash chain corresponds with the merkle root.
3. Insert/Remove the node and propagate new hashes up the chain to the root.
4. Update the root in the CFM server.

Step 1 validates the state of the Merkle tree in memory (which may have been corrupted by an attacker). Step 2 validates the memory location and value of a pointer.

Its important to note that the computation of these steps cannot be dependent on the state of the program's memory before step 1 and must be free of memory errors. If this is the case then the control flow integrity of the entire program can be guaranteed.

4.2 CFM Server

A Control Flow Manager (CFM) was implemented and set up as a minix service with a static process ID. We needed this server because have access to any assembler instructions that could do AES encryption or store the merkle root within a register. There were three different methods of communication that we considered before doing the implementation.

4.2.1 Driver communication

One way of communicating with the server is to create a driver that is configured to be allowed to communicate with the CFM. This solution uses a character-based stream of data where we would check for different keywords, followed by some parameters. The problem with this solution is that it is a lot of work to fill up a buffer, check that buffer continuously, serialize the parameters, parse the parameters to their original type and then make the respective function call to the CFM.

4.2.2 Proxy service

Using an existing server such as the Process Manager (PM) is another way to relay IPC messages from the user-space program to the PM to the CFM. This way, we do not need to add any new service to the system configuration since the PM already accepts syscalls from user-space programs. The benefit of using this solution is the added safety it provides, because within the proxy we can make additional checks and guards which the PM could store and handle. This way we make sure that the process or control flow is not being manipulated before actually sending the syscall to the CFM.

4.2.3 Direct communication

We ended up using direct communication with the service through IPC from the user-space program, by using a wrapper that does a syscall and where the CFM is added to the system configuration. This was the easiest and most clean solution since we did not need to set up the syscalls twice for both the PM and CFM, as well as match the parameters and naming conventions between them. We could simply add the user-space macro which in turn use the merkle library that could efficiently update or retrieve the merkle root.

4.3 User space macros

We used C-macros as a temporary solution to protect pointers and return addresses, they are added manually to the user-space program, but this is of course something that we would like to do at compile time instead automatically. However, they provide a proof of concept that our solution works and can be inserted as a guard to detect buffer overflows. Pointers are protected by storing it right after it is initialized, then before a pointer is used, i.e. assigned a new value, we validate that it is the same, see Figure 4

```
state.x = malloc(sizeof(int) * 1);
STORE_POINTER(state.x)
printf("addresses %p %p %p %p \n", &pwd_diff, &(state.x), state.x, state.buffer);

strcpy(state.buffer, argv[2]);
printf("addresses %p %p %p %p \n", &pwd_diff, &(state.x), state.x, state.buffer);
VALIDATE_POINTER(state.x)
*(state.x) = atoi(argv[3]);
```

Figure 4: Macro protection of a pointer

A return address is protected by storing it when a function is called, then validating it right before the function returns, see Figure 5

4.4 AES

Instead of taking advantage of special AES instructions a simple AES library was added and an AES function call was added in the CFM server. The function takes an arbitrary length as input and returns a hash of fixed length, creating a keyed hash.

```
void foo(char *input) {
    STORE_POINTER(__builtin_return_address(0))

    char buffer[8];
    strcpy(buffer, input);
    printf("My name is %s\n", buffer);

    VALIDATE_POINTER(__builtin_return_address(0))
    return;
}
```

Figure 5: Macro protection of a return address

5 Project status

There would be a couple of things on the todo-list if this project were to be continued. First of all, we would want the implementation of the Merkle tree to be changed so that it is self-balancing. This will drastically reduce the worst-case traverse time and therefore shorten the time needed to validate a pointer, thereby increasing the performance. Another idea is to modify the tree to handle arbitrary data instead of just pointers, making it a lot more versatile in its use. Doing this would improve the overall security that is provided. Another step that would improve security is if it was possible to monitor multiple processes at the same time. Currently, our CFM can only store a single Merkle tree root and, therefore, only handle one process at a time. This is because we can't have dynamically amount of memory allocated. Clearly, this reduces the usability of the server.

A long-time goal with this design, if this was to be implemented for real, is to have this moved and implemented in hardware instead. This would be possible because of AES-NI, which is an AES instruction set integrated into many processors.

6 Contribution

Firstly I contributed by setting up the communication between the user-space program and CFM, as well as setting up the skeleton and configuration for the CFM service. Making sure that the overarching execution flow of our solution was working from triggering the macro, subsection 4.3, making the syscall and returning data back from the CFM. The CFM was set up adding/updating the following files:

- CFM header to minix distribution - *distrib/sets/lists/minix-comp/mi*
- CFM boot process to minix distribution - *distrib/sets/lists/minix-kernel/mi*
- CFM service configuration and direct communication - *etc/system.conf*
- Wrapper header - *minix/include/minix/cfm.h*
- Include wrapper header in build - *minix/include/minix/Makefile*
- Wrapper source - *minix/lib/libsys/cfm.c*
- Include wrapper source in libsys build - *minix/lib/libsys/Makefile*
- Static PID for CFM service and syscall addresses - *minix/include/minix/com.h*
- Include CFM in boot process table - *minix/servers/rs/table.c*
- CFM server source - *minix/src/minix/servers/cfm/main.c*
- CFM server store - *minix/src/minix/servers/cfm/store.c*

- CFM server prototype header - *minix/servers/cfm/proto.h*
- CFM server Makefile - *minix/servers/cfm/Makefile*
- Include CFM server in build - *minix/servers/Makefile*
- Set boot load order - *releasetools/Makefile*
- IPC message struct - *minix/include/minix/ipc.h*

The *main.c* file provides an infinite loop which constantly waits for incoming messages, it will check the callnr contained in the message as *m_type*, corresponding to some syscall defined in *com.h*. Then via a switch statement the appropriate action will be taken. The actions are defined in *store.c*, these functions typically modify the message and return some response value, such as EINVAL, invalid request, EDONTREPLY, do not reply or OK if the action was successful. Finally the message is sent back to the caller, also contained in the message, as *m_source* which corresponds to the process number of the caller.

I provided a small example of sending data back and forth between the user-space program and CFM, with data of arbitrary length. The IPC message struct uses a single buffer so that it can be used bidirectionally, see Figure 6. Fortunately, we could also fit the whole merkle tree root within a single message with a 52 byte buffer, so we did not have to use magic grants, even though this is something we also investigated.

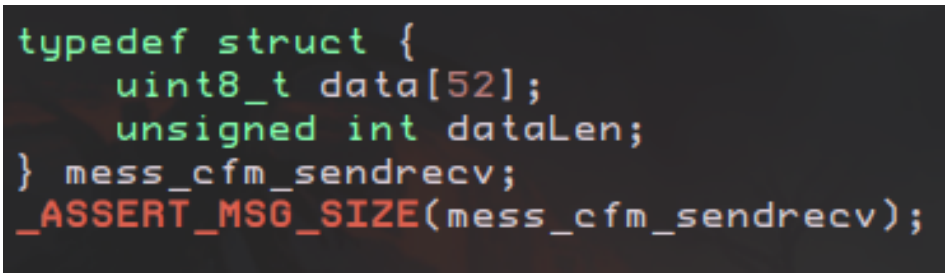

Figure 6: IPC message for the CFM

Apart from the CFM, I was focusing on the buffer overflow from the attacker's perspective, setting up the exploit, and demonstrating the vulnerabilities in detail of what happens in memory and registers when the buffer is overflowed. I also researched the current protection mechanism standards such as stack canaries and ASLR and how they can be circumvented explained in section 2. Some of the exploit test code is present in the cfm_client branch under minix/src/games/cfm_test. I was testing the attacks on linux with python *exploit.py* and gdb for efficiency. To disable ASLR on linux I used the command

```
sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

And in order to disable stack canaries I used the -fno-stack-protector flag in gcc

```
gcc cfm_test.c -fno-stack-protector -g -o cfm_test &&
sudo gdb --args ./cfm_test $(python3 exploit.py)
```

With gdb I was able to find the address of *print_pwd* in *cfm_test.c* which is the target address of the attack that I overwrite the return address of the stackframe of foo to. The main problem was that functions such as *strlen* and *strcpy* stops when encountering a zero byte "x00" which was happening quite often for*print_pwd*. The only way to circumvent it is by abusing some other vulnerable function call such as an incorrect data length used in *memcpy* or *atoi* or targeting addresses without any zero byte.

I was also investigating whether it would be possible to inject assembly code into a buffer and then use a nop sled to trigger malicious code from the program, *exploit_sled.py*. The nop sled works by executing nop instructions "x90" until eventually, it reaches the malicious code. This is advantageous because the return

address should now point to a location in the stack (7fffff...) rather than the text section (00005555...) which does not have any zero byte.

Similarly I demonstrated with gdb the exact memory manipulation step by step of the pointer overflow in *minix/src/games/overflow*.

References

[1] Ali Jose Mashtizadeh et al. *Cryptographically Enforced Control Flow Integrity*. 2014. DOI: 10.48550/ARXIV.1408.1451. URL: https://arxiv.org/abs/1408.1451.

[2] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.