4. Mitigating Memory-Safety Vulnerabilities

4. Mitigating Memory-Safety Vulnerabilities

4.1. Use a memory-safe language

Some modern languages are designed to be intrinsically memory-safe, no matter what the programmer does. Java, Python, Go, Rust, Swift, and many other programming languages include a combination of compile-time and runtime checks that prevent memory errors from occurring. Using a memory safe language is the *only* way to stop 100% of memory safety vulnerabilities. In an ideal world, everyone would program in memory-safe languages and buffer overflow vulnerabilities would no longer exist. However, because of legacy code and perceived[1] performance concerns, memory-unsafe languages such as C are still prevalent today.

4.2. Writing memory-safe code

One way to ensure memory safety is to carefully reason about memory accesses in your code, by defining pre-conditions and post-conditions for every function you write and using invariants to prove that these conditions are satisfied. Although it is a good skill to have, this process is painstakingly tedious and rarely used in practice, so it is no longer in scope for this class. If you'd like to learn more, see this lecture from David Wagner: video, slides.

Another example of defending against memory safety vulnerabilities is writing memory-safe code through defensive programming and using safe libraries. Defensive programming is very similar to defining pre and post conditions for every written function, wherein you always add checks in your code just in case something could go wrong. For example, you would always check that a pointer is not null before dereferencing it, even if you are sure that the pointer is always going to be valid. However, as mentioned earlier, this relies a lot on programmer discipline and is very tedious to properly implement. As such, a more common method is to use safe libraries, which, in turn, use functions that check bounds so you don't have to. For example, using `fgets` instead of `gets`, `strncpy` or `strlcpy` instead of `strcpy`, and `snprintf` instead of `sprintf`, are all steps towards making your code slightly more safe.

4.3. Building secure software

Yet another way to defend your code is to use tools to analyze and patch insecure code. Utilizing run-time checks that do automatic bound-checking, for example is an excellent way to help your code stay safe. If your check fails, you can direct it towards a controlled crash, ensuring that the attacker does not succeed. Hiring someone to look over your code for memory safety errors, though expensive, can prove to be extremely beneficial as well. You can also probe your own system for vulnerabilities, by subjecting your code to thorough tests. Fuzz testing, or testing with random inputs, testing corner cases, and using tools like Valgrind (to detect memory leaks), are all excellent ways to help test your code. Though it is pretty difficult to know whether you have tested your code "enough" to deem it safe, there are several code-coverage tools that can help you out.

---

[1]The one real performance advantage C has over a garbage collected language like Go is a far more deterministic behavior for memory allocation. But with languages like Rust, which are safe but not garbage collected, this is no longer an advantage for C.

4.4. Exploit mitigations

Sometimes you might be forced to program in a memory-unsafe language, and you cannot reason about every memory access in your code. For example, you might be asked to update an existing C codebase that is so large that you cannot go through and reason about every memory access. In these situations, a good strategy is to compile and run code with *code hardening defenses* to make common exploits more difficult.

Code hardening defenses are *mitigations*: they try to make common exploits harder and cause exploits to crash instead of succeeding, but they are not foolproof. The only way to prevent *all* memory safety exploits is to use a memory-safe language. Instead, these mitigations are best thought of as defense-in-depth: they cannot prevent all attacks, but by including many different defenses in your code, you can prevent more attacks. Over the years, there has been a back-and-forth arms race between security researchers developing new defenses and attackers developing new ways to subvert those defenses.

The rest of this section goes into more detail about some commonly-used code hardening defenses, and techniques for subverting those defenses. In many cases, using multiple mitigations produces a synergistic effect: one mitigation on its own can be bypassed, but a combination of multiple mitigations forces an attacker to discover multiple vulnerabilities in the target program.

4.5. Mitigation: Non-executable pages

Many common buffer overflow exploits involve the attacker writing some machine code into memory, and then redirecting the program to execute that injected code. For example, one of the stack smashing attacks in the previous section (`[shellcode] + [4 bytes of garbage] + [address of buf]`) involves the attacker writing machine code into memory and overwriting the rip to cause the program to execute that code.

One way to defend against this category of attacks is to make some portions of memory *non-executable*. What this means is that the computer should not interpret any data in these regions as CPU instructions. You can also think of it as not allowing the eip to ever contain the address of a non-executable part of memory.

Modern systems separate memory into *pages* in order to support virtual memory (see 61C or 162 to learn more). To defend against memory safety exploits, each page of memory is set to either be *writable or executable, but not both*. If the user can write to a page in memory, then that page of memory cannot be interpreted as machine instructions. If the program can execute a page of memory as machine instructions, then the user cannot write to that page.

This defense stops the stack smashing attack in the previous section where the attacker wrote machine code into memory. Because the attacker wrote machine code to a page in memory, that page cannot be executed as machine instructions, so the attack no longer works.

This defense has several names in practice, including WˆX (Write XOR Execute), DEP (Data Execution Prevention), and the NX bit (no-execute bit).

4.6. Subverting non-executable pages: Return into libc

Non-executable pages do not stop an attacker from executing existing code in memory. Most C programs import libraries with thousands or even million lines of instructions. All of these instructions are marked as executable (and non-writable), since the programmer may want to call these functions legitimately.

An attacker can exploit this by overwriting the rip with the address of a C library function. For example, the `execv` function lets the attacker start executing the instructions of some other executable.

Some of these library functions may take arguments. For example, `execv` takes a string with the filename of the program to execute. Recall that in x86, arguments are passed on the stack. This means that an attacker can carefully place the desired arguments to the library function in the right place on the stack, so that when the library function starts to execute, it will look on the stack for arguments and find the malicious argument placed there by the attacker. The argument is not being run as code, so non-executable pages will not stop this attack.

4.7. Subverting non-executable pages: Return-oriented programming

We can take this idea of returning to already-loaded code and extend it further to now execute arbitrary code. Return-oriented programming is a technique that overwrites a chain of return addresses starting at the RIP in order to execute a series of "ROP gadgets" which are equivalent to the desired malicious code. Essentially, we are constructing a custom shellcode using pieces of code that already exist in memory. Instead of executing an existing function, like we did in "Return to libc", with ROP you can execute your own code by simply executing different pieces of different code. For example, imagine we want to add 4 to the value currently in the EDX register as part of a larger program. In loaded memory, we have the following functions:

```
foo:
    ...
    0x4005a1 <foo+33> mov %edx, %eax
    0x4005a3 <foo+35> leave
    0x4005a4 <foo+36> ret
    ...
bar:
    ...
    0x400604 <bar+20> add $0x4, %eax
    0x400608 <bar+24> pop %ebx
    0x40060a <bar+26> leave
    0x40060b <bar+27> ret
```

To emulate the `add $0x4, %edx` instruction, we could move the value in EDX to EAX using the gadget in `foo` and then add 4 to EAX using the gadget in `bar`! If we set the first return address to `0x004005a1` and the second return address to `0x00400604`, we produce the desired result. Each time we jump to ROP gadget, we eventually execute the `ret` instruction and then pop the next return address off the stack, jumping to the next gadget. We just have to keep track that our desired value is now in a different register, and because we execute a `pop %ebx` instruction in `bar` before we return, we also have to remember that the value in EBX has been updated after executing these gadgets—but these are all behaviors that we can account for using standard compiler techniques. In fact, so-called "ROP compilers" exist to take an existing vulnerable program and a desired execution flow and generate a series of return addresses.

The general strategy for executing ROPs is to write a chain of return addresses at the RIP to achieve the behavior that we want. Each return address should point to a gadget, which is a small set of assembly instructions that already exist in memory and usually end in a `ret` instruction (note that gadgets are not functions, they don't need to start with a prologue or end with an epilogue!). The gadget then executes its instructions and ends with a `ret` instruction, which tells the code to jump to the next address on the stack, thus allowing us to jump to the next gadget!

If the code base is big enough, meaning that the code imports enough libraries, there are usually enough gadgets in memory for you to be able to run any shellcode that you want. In fact, ROP compilers exist on the Internet that will automatically generate an ROP chain for you based on a target binary and desired malicious code! ROP has become so common that non-executable pages are no longer a huge issue for attackers nowadays; while having writable and executable pages makes an attacker's life easier, not a lot of effort has to be put in to subvert this defense mechanism.

4.8. Mitigation: Stack canaries

In the old days, miners would protect themselves against toxic gas buildup in the mine by bringing a caged canary into the mine. These particularly noisy birds are also sensitive to toxic gas. If toxic gas builds up in the mine, the canary dies first, which gives the miners a warning sign that the air is toxic and they should evacuate immediately. The canary in the coal mine is a sacrificial animal: the miners don't expect it to survive, but its death acts as a warning to save the lives of the miners.

We can use this same idea to prevent against buffer overflow attacks. When we call a function, the compiler places a known dummy value, the *stack canary*, on the stack. This canary value is not used by the function

at all, so it should stay unchanged throughout the duration of the function. When the function returns, the compiler checks that the canary value has not been changed. If the canary value has changed, then just like the canary in the mine dying, this is evidence that something bad has happened, and the program will crash before any further damage is done.

Like the canary in the coal mine, the stack canary is a sacrifical value: it has no purpose in the function execution and nothing bad happens if it is changed, but the canary changing acts as a warning that someone may be trying to exploit our program. This warning lets us safely crash the program instead of allowing the exploit to succeed.

The stack canary uses the fact that many common stack smashing attacks involve overflowing a local variable to overwrite the saved registers (sfp and rip) directly above. These attacks often write to *consecutive, increasing* addresses in memory, without any gaps. In other words, if the attacker starts writing at a buffer and wants to overwrite the rip, they must overwrite everything in between the buffer and the rip.

The stack canary is placed directly above the local variables and directly below the saved registers (sfp and rip):

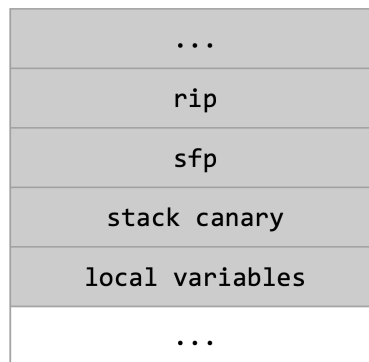| ... |
| rip |
| sfp |
| stack canary |
| local variables |
| ... |

Figure 1: A stack canary located between the sfp and the local variables of a given stack frame

Suppose an attacker wants to overflow a local variable to overwrite the rip on the stack, and the vulnerability only allows the attacker to write to consecutive, increasing addresses in memory. Then the attacker must overwrite the stack canary before overwriting the rip, since the rip is located above the buffer in the stack.

Before the function returns and starts executing instructions at the rip, the compiler will check whether the canary value is unchanged. If the attacker has attempted to overwrite the rip, they will have also changed the canary value. The program will conclude that something bad is happening and crash before the attacker can take control. Note that the stack canary detects an attack before the function returns.

The stack canary is a random value generated at *runtime*. The canary is 1 word long, so it is 32 bits long in 32-bit architectures. In Project 1, the canary is 32 completely random bits. However, in reality, stack canaries are usually guaranteed to contain a null byte (usually as the first byte). This lets the canary defend against string-based memory safety exploits, such as vulnerable calls to `strcpy` that read or write values from the stack until they encounter a null byte. The null byte in the canary stops the `strcpy` call before it can copy past the canary and affect the rip.

The canary value changes each time the program is run. If the canary was the same value each time the program was run, then the attacker could run the program once, write down the canary value, then run the program again and overwrite the canary with the correct value. Within a single run of the program, the canary value is usually the same for each function on the stack.

Modern compilers automatically add stack canary checking when compiling C code. The performance overhead from checking stack canaries is negligible, and they defend against many of the most common exploits, so there is really no reason not to include stack canaries when programming in a memory-unsafe language.

4.9. Subverting stack canaries

Stack canaries make buffer overflow attacks harder for an attacker, but they do not defend programs against all buffer overflow attacks. There are many exploits that the stack canary cannot detect:

- Stack canaries can't defend against attacks outside of the stack. For example, stack canaries do nothing to protect vulnerable heap memory.
- Stack canaries don't stop an attacker from overwriting other local variables. Consider the `authenticated` example from the previous section. An attacker overflowing a buffer to overwrite the `authenticated` variable never actually changes the canary value.
- Some exploits can write to non-consecutive parts of memory. For example, format string vulnerabilities let an attacker write directly to the rip without having to overwrite everything between a local variable and the rip. This lets the attacker write "around" the canary and overwrite the rip without changing the value of the canary.

Additionally, there are several techniques for defeating the stack canary. These usually involve the attacker modifying their exploit to overwrite the canary with its original value. When the program returns, it will see that the canary is unchanged, and the program won't detect the exploit.

Guess the canary: On a 32-bit architecture, the stack canary usually only has 24 bits of entropy (randomness), because one of the four bytes is always a null byte. If the attacker runs the program with an exploit, there is a roughly 1 in 2^{24} chance that the the value the attacker is overwriting the canary with matches the actual canary value. Although the probability of success is low on one try, the attacker can simply run the program 2^{24} times and successfully exploit the program at least once with high probability.

Depending on the setting, it may be easy or hard to run a program and inject an exploit 2^{24} times. If each try takes 1 second, the attacker would need to try for over 100 days before they succeed. If the program is configured to take exponentially longer to run each time the attacker crashes it, the attacker might never be able to try enough times to succeed. However, if the attacker can try thousands of times per second, then the attacker will probably succeed in just a few hours.

On a 64-bit architecture, the stack canary has 56 bits of randomness, so it is significantly harder to guess the canary value. Even at 1,000 tries per second, an attacker would need over 2 million years on average to guess the canary!

Leak the canary: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. An attacker could use this vulnerability to leak the value of the canary, write it down, and then inject an exploit that overwrites the canary with its leaked value. All of this can happen within a single run of the program, so the canary value doesn't change on program restart.

4.10. Mitigation: Pointer authentication

As we saw earlier, stack canaries help detect if an attacker has modified the rip or sfp pointers by storing a secret value on the stack and checking if the secret value has been modified. As it turns out, we can generalize this idea of using secrets on the stack to detect when an attacker modifies *any* pointer on the stack.

Pointer authentication takes advantage of the fact that in a 64-bit architecture, many bits of the address are unused. A 64-bit address space can support 2^{64} bytes, or 18 exabytes of memory, but we are a long way off from having a machine with this much memory. A modern CPU might support a 4 terabyte address space, which means 42 bits are needed to address all of memory. This still leaves 22 unused bits in every address and pointer (the top 22 bits in the address are always 0).

Consider using these unused bits to store a secret like the stack canary. Any time we need to store an address on the stack, the CPU first replaces the 22 unused bits with some secret value, known as the *pointer authentication code*, or PAC, before pushing the value on the stack. When the CPU reads an address off the stack, it will check that the PAC is unchanged. If the PAC is unchanged, then the CPU replaces the PAC with the original unused bits and uses the address normally. However, if the PAC has been changed, this is a

warning sign that the attacker has overwritten the address! The CPU notices this and safely crashes the program.

As an example, suppose the rip of a function in a 64-bit system is `0x0000001234567899`. The address space for this architecture is 40 bits, which means the top 24 bits (3 bytes) are always 0 for every address. Instead of pushing this address directly on the stack, the CPU will first replace the 3 unused bytes with a PAC. For example, if the PAC is `0xABCDEF`, then the address pushed on the stack is `0xABCDEF1234567899`.

This address (with the secret value inserted) is invalid, and dereferencing it will cause the program to crash. When the function returns and the program needs to start executing instructions at the rip, the CPU will read this address from the stack and check that the PAC `0xABCDEF` is unchanged. If the PAC is correct, then the CPU replaces the secret with the original unused bits to make the address valid again. Now the CPU can start executing instructions at the original rip `0x0000001234567899`.

Now, an attacker trying to overwrite the rip would need to know the PAC in order to overwrite the rip with the address of some attacker shellcode. If the attacker overwrites the PAC with an incorrect value, the CPU will detect this and crash the program.

We can strengthen this defense even further. Since it is the CPU's job to add and check the PAC, we can ask the CPU to use a different PAC for every pointer stored on the stack. However, we don't want to store all these PACs on the CPU, so we'll use some special math to help us generate secure PACs on the fly.

Consider a special function $f(\mathsf{KEY}, \mathsf{ADDRESS})$. The function f takes a secret key KEY and an address $\mathsf{ADDRESS}$, and outputs a PAC by performing some operation on these two inputs. This function is deterministic, which means if we supply the same key and address twice, it will output the same secret value twice. This function is also secure: an attacker who doesn't know the value of KEY cannot output secret values of their own.[2]

Now, instead of using the same PAC for every address, we can generate a different PAC for each address we store in memory. Every time an address needs to be stored in memory, the CPU runs f with the secret key and the address to generate a unique secret value. Every time an address from memory needs to be dereferenced, the CPU runs f again with the secret key and the address to re-generate the PAC, and checks that the generated value matches the value in memory. The CPU only has to remember the secret key, because all the secret values can be re-generated by running f again with the key and the address.

Using a different PAC for every address makes this defense extremely strong. An attacker who can write to random parts of memory can defeat the stack canary, but cannot easily defeat pointer authentication: they could try to leave the PAC untouched, but because they've changed the address, the old secret value will no longer check out. The CPU will run f on the attacker-generated address, and the output will be different from the old secret value (which was generated by running f on the original address). The attacker also cannot generate the correct secret value for their malicious address, because they don't know what the secret key is. Finally, an attacker could try to leak some addresses and secret values from memory, but knowing the PACs doesn't help the attacker generate a valid PAC for their chosen malicious address.

With pointer authentication enabled, an attacker is never able to overwrite pointers on the stack (including the rip) without generating the corresponding secret for the attacker's malicious address. Without knowing the key, the attacker is forced to guess the correct secret value for their address. For a 20-bit secret, the attacker has a 1 in 2^{20} chance of success.

Another way to subvert pointer authentication is to find a separate vulnerability in the program that allows the attacker to trick the program into creating a validated pointer. The attacker could also try to discover the secret key stored in the CPU, or find a way to subvert the function f used to generate the secret values.

4.11. Mitigation: Address Space Layout Randomization (ASLR)

Recall the stack smashing attacks from the previous section, where we overwrote the rip with the address of some malicious code in memory. This required knowing the exact address of the start of the malicious code.

---

[2]This function is called a MAC (message authentication code), and we will study it in more detail in the cryptography unit.

ASLR is a mitigation that tries to make predicting addresses in memory more difficult.

Although we showed that C memory is traditionally arranged with the code section starting at the lowest address and the stack section starting at the highest address, nothing is stopping us from shifting or rearranging the memory layout. With ASLR, each time the program is run, the beginning of each section of memory is randomly chosen. Also, if the program imports libraries, we can also randomize the starting addresses of each library's source code.

ASLR causes the absolute addresses of variables, saved registers (sfp and rip), and code instructions to be different each time the program is run. This means the attacker can no longer overwrite some part of memory (such as the rip) with a constant address. Instead, the attacker has to guess the address of their malicious instructions. Since ASLR can shuffle all four segments of memory, theoretically, certain attacks can be mitigated. By randomizing the stack, the attacker cannot place shellcode on the stack without knowing the address of the stack. By randomizing the heap, the attacker, similarly, cannot place shellcode on the heap without knowing the address of the heap. Finally, by randomizing the code, the attacker cannot construct an ROP chain or a return-to-libc attack without knowing the address of the code.

There are some constraints to randomizing the sections of memory. For example, segments usually need to start at a page boundary. In other words, the starting address of each section of memory needs to be a multiple of the page size (typically 4096 bytes in a 32-bit architecture).

Modern systems can usually implement ASLR with minimal overhead because they dynamically link libraries at runtime, which requires each segment of memory to be relocatable.

4.12. Subverting ASLR

The two main ways to subvert ASLR are similar to the main ways to subvert the stack canary: guess the address, or leak the address.

Guess the address: Because of the constraints on address randomization, a 32-bit system will sometimes only have around 16 bits of entropy for address randomization. In other words, the attacker can guess the correct address with a 1 in 2^{16} probability, or the attacker can try the exploit 2^{16} times and expect to succeed at least once. This is less of a problem on 64-bit systems, which have more entropy available for address randomization.

Like guessing the stack canary, the feasibility of guessing addresses in ASLR depends on the attack setting. For example, if each try takes 1 second, then the attacker can make 2^{16} attempts in less than a day. However, if each try after a crash takes exponentially longer, 2^{16} attempts may become infeasible.

Leak the address: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. The stack often stores absolute addresses, such as pointers and saved registers (sfp and rip). If the attacker can leak an absolute address, they may be able to determine the absolute address of other parts of memory relative to the absolute address they leaked.

Note that ASLR randomizes absolute addresses by changing the start of sections of memory, but it does not randomize the *relative* addresses of variables. For example, even if ASLR is enabled, the rip will still be 4 bytes above the sfp in a function stack frame. This means that an attacker who leaks the absolute address of the sfp could deduce the address of the rip (and possibly other values on the stack).

4.13. Combining Mitigations

We can use multiple mitigations together to force the attacker to find multiple vulnerabilities to exploit the program; this is a process known as *synergistic protection*, where one mitigation helps strengthen another mitigation. For example, combining ASLR and non-executable pages results in an attacker not being able to write their own shellcode, because of non-executable pages, and not being able to use existing code in memory, because they don't know the addresses of that code (ASLR). Thus, to defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities. First, they need to find a way to leak memory and reveal the

address location (to defeat ASLR). Next, they need to find a way to write to memory and write an ROP chain (to defeat non-executable pages).