Runtime Defenses
6.1600 Course Staff
Fall 2023

So far, we have discussed ways to achieve security in the presence of the reality that code has bugs by separating our system into components with limited privilege and by attempting to find bugs in our code. However, we will still be left with some bugs. In this chapter, we will aim to monitor execution of the system to detect and buggy behavior and halt in its presence.

There is no cut-and-dry method to achieve this. However, we can think about certain attacks that we might care about and construct defenses to make those attacks harder.

1 Buffer Overflow Defenses

We have discussed the pervasive buffer overflow attack, which takes advantage of a missing bounds check to overwrite memory beyond the bounds of an array, often modifying the current function’s return address to cause the attacked system to run attacker-specified code which is placed in the buffer itself.

1.1 Non-Executable Stack

Programming languages like C use a region of memory called the stack to store function variables. The input buffer in the buffer overflow attack is typically placed in this stack memory region. However, program code is typically in a separate region of memory. CPUs also allow us to add permissions to different memory regions: we can mark each as read (R), write (W), and/or execute (X). To minimize the possible effects of a buffer overflow attack, one approach is to prevent running code from the stack at all by marking the stack as RW only.

This seems to eliminate a major piece of an effective buffer overflow attack: the attacker can no longer supply code of their choice and point to it with the return address. However, modern attackers have worked around this with something called return-oriented programming: with the ability to supply their own code removed, attackers must find code that already exists to do what they like. This may be full existing functions, but more likely attackers will set the return address to point into the middle of some function and execute just a fragment that does something useful. It turns out that with more work, it is possible to perform many attacks using only code
that already exists on the system.

1.2 Stack Canary

To try to remove the adversary’s ability to overwrite the return address in the presence of a buffer overflow, another defense is to insert a stack canary in every stack frame between the function variables and the return address. At the start of each function an operation is inserted to write this canary to some value. At the end of the function before returning, a check on the canary value will be inserted: if the canary has changed, something must have overwritten it and the program should exit to avoid running unknown code.

This is effective because in a buffer overflow scenario, the attacker needs to write memory sequentially until the address they care about writing is reached: if the canary is between the function variables and the return address, the attacker must overwrite the canary to modify the return address. However, this is not a perfect defense: if the attacker writes the same value to the canary as was already there, it will go undetected. Therefore, the canary value must be hard for the attacker to guess. Using a random value seems promising, and in order to avoid the performance overhead of picking a new random value on every function invocation, a canary value is often picked at program startup and stored outside of the stack to prevent a buffer overflow from overwriting the reference canary value.

This defense is still not perfect—for example, it does not prevent an attacker from overwriting function pointers. However, it does make a successful attack significantly harder.

1.3 Address Space Layout Randomization (ASLR)

Another approach for these buffer overflow-style attacks is to make it hard for the adversary to guess a useful address to jump to. To do this, many modern systems randomize the locations of code, stack, and heap memory regions when a process starts. With this defense in place, an attacker needs to learn the location of the code memory region in order to do any meaningful return-oriented programming attack.

1.4 Bounds Checking with Fat Pointers

All of the defenses so far have attempted only to minimize the damage of a buffer overflow after it has been exploited. However, we could prevent a more comprehensive suite of attacks if we could make sure that our code never reads or writes a pointer that is outside the bounds of a given buffer. Memory-safe languages like Go,
Rust, or Python have this bounds checking built in, but attempting to retrofit a C compiler to achieve this bounds-checking presents additional challenges.

In the Fat Pointers technique, pointers are augmented to include the base and the limit of the buffer the pointer belongs to. This base and limit are initialized on an allocation, and on a dereference a check is inserted to guarantee that the current value of the pointer is within the region specified by the base and limit. Pointer arithmetic preserves the base and limit but modifies the pointer itself as before, allowing the pointer to possibly go out of bounds.

Unlike a nonexecutable stack, stack canaries, and ASLR, fat pointers are not widely used. This is largely because the modified “fat” pointers are too incompatible with existing C code. C code sometimes casts pointers to integers and back again, causing issues for our new 24-byte pointers, and the modified pointer size can modify memory layout in ways that the code does not expect.

1.5 Control Flow Integrity (CFI)

Another approach is to target jumps specifically: we know that there is a certain set of valid jumps in our program: code should only be jumping to return points in functions and function starts. If we could detect when a function returns by jumping to some target outside of this set, we can prevent many attacks where an attacker causes the program to jump to code that can be used maliciously. Checking these jump targets is called Control Flow Integrity.

To implement CFI, we need to add several checks on different kinds of jumps. For direct jumps, we know at compile time that these jumps are valid since there is nothing that the attacker can control. No checks are needed. For computed jumps, that use some variable in the jump target, we need to check that the return address is a valid jump target. To do this, compilers that implement CFI create a bitmap that maps a hash of each address to a bit that indicates whether that address is a valid jump target. This allows achieving good performance, but there is still necessarily an overhead. Many compilers provide CFI as a compilation flag.

2 Input Sanitization Defenses

We have discussed SQL injection and cross-site scripting, which allowed an attacker to insert code by adding meaningful characters like " or > into their input. To protect against this, user inputs should be sanitized, a process that takes these special characters and escapes them, converting them to some other sequence so that they are
treated as plain text.

2.1 Taint Tracking

To check for failures in this sanitization at runtime, a common approach is called **taint tracking**. Input is marked as **tainted** at sources of user input. At functions that perform this escaping, this taint label is removed. Sensitive functions such as the HTML renderer are marked as **sinks**, and any time a sink is run a check will be inserted to ensure that the input data is not tainted. If the data is tainted, some part of the input must not have been sanitized since it came from the user, and the program should halt to avoid an exploit.

This is implemented in many browsers to prevent cross-site scripting using **Trusted Types**: for JavaScript calls that update the displayed HTML, such as `innerHTML = foo`, browsers may restrict the type of `foo` to ensure that the code explicitly converts its type to something like `TrustedHTML`. This does not guarantee that the sanitization was done correctly, but does ensure that the programmer acknowledged the risk in their code.