Runtime Defenses

6.1600 Course Staff

Fall 2023

We have considered a number of ways to improve software security. First, we explored *privilege separation* as a way to architect a system so that bugs do not lead to catastrophic security failures. Next, we looked at *bug finding*—techniques to find and eliminate bugs in source code before we use the code in production. Now, we will discuss *runtime defenses*: how to detect buggy behavior as it occurs in a piece of production software so that we can halt the program when a bug occurs.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

There are a number of reasons why it is difficult to build runtime defenses. We must:

- determine the classes of bugs that we want to find,
- identify which components of the system that we want to monitor,
- figure out how to avoid *false positives* (erroneously halting a program when there is no bug), and
- try to implement these runtime defenses with minimal overhead, since we will apply these defenses to production code.

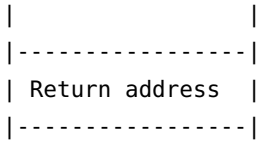
1 *Defenses against buffer overflows*

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.

An example of C code that is vulnerable to a buffer-overflow attack is this:

```
1 void f() {
2   char buf[128];
3   gets(buf);
4 }
```

The call-stack layout for this program will look like this:



```
| buf[127]        |
| ....            |
| buf[1]          |
| buf[0]          |
|-----------------|
|                 |
|                 |
|     ....        |
|-----------------|
```

The `gets` function in C will read from `stdin` until it finds a NULL (`\0`) character in the input string. If the string has length > 127, the `gets` function could copy adversarially generated input byte onto the stack, overwriting the return address of the function `f`.

There are three steps involved in executing a buffer-overflow attack:

1. write past the end of a buffer,
2. cause the victim process to jump to an adversary-controlled address, and
3. cause the victim process to run adversarial code.

Runtime defenses against buffer overflows can try to disrupt each of these three steps.

1.1 Non-Executable Stack

A first defense against buffer-overflow attacks is to prevent the CPU from executing code on the C call stack. In traditional buffer-overflow attacks, an attacker will somehow place some adversarial code on the stack (e.g., in the buffer `buf` in the example above) and then cause the victim process to jump to that code on the stack.

However, C usually puts normal program code in a separate region of memory—not on the stack. Modern CPUs allow us to add permissions to different memory regions: the OS can mark each region as read (R), write (W), and/or execute (X). To make buffer-overflow attacks more difficult, we can prevent the CPU from running code from the stack at all by marking the stack as RW only.

This is sometimes called an NX defense—for "no execute."

Marking the stack as non-executable 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 in a victim process.

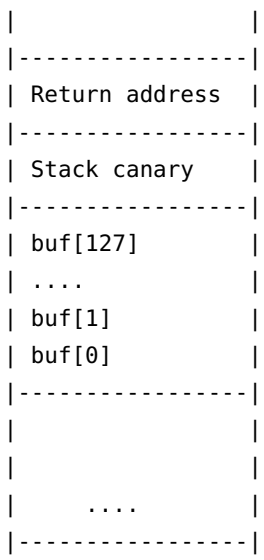
1.2 Stack Canary

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. A stack canary is typically a secret random value, chosen when the program starts running.

A buffer-overflow attack does not directly allow the attacker to read arbitrary memory, so the attacker has no direct way to read the canary before overflowing the buffer.

At the start of each function, the compiler inserts instruction that write this canary to some value. At the end of the function before returning, the compiler also adds some code that checks that the canary value has not changed. If the canary has changed, the attacker must have overflowed a buffer and the program should exit to avoid running unknown code.

A clever non-random canary includes a collection of the string-terminating characters, such as `0` `n` `r`. Many C functions that read strings from input, such as `gets` will stop reading once they reach one of these values, so it could be difficult for an attacker to feed in an overflowing string that includes these characters.



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.

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.

There are a few ways to subvert canaries:

- An attacker can corrupt *data* on the stack, even if it does not corrupt the return address. This could be very bad for the program's behavior.
- An attacker might find a way to read the canary from memory (e.g., if there is some other bug in the program) and then execute the traditional buffer-overflow attack to overwrite the return pointer.
- In a forking web server, the child process may have the same canary value as the parent process. An attacker can potentially exploit this to learn the canary (See Andrea Bittau et al. "Hacking blind". In: *IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 227–242).

1.3 Address Space Layout Randomization (ASLR)

Another approach to defend against buffer overflow-style attacks is to make it more difficult 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 mount a return-oriented programming attack (Section 1.1).

Implementing ASLR requires compiler support. Most modern compilers do.

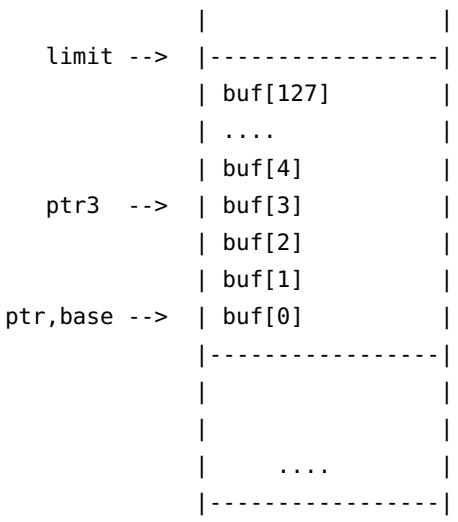
A weakness of ASLR schemes is that they typically shift the location of an entire region of memory: the entire heap, stack, and code sections move around in memory, but the layout within each section is typically fixed at compile time. Thus if the adversary can learn the location in memory of the code for a single function, it can mount an effective 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 an attacker has exploited it. 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 such as Go, Rust, and Python have this bounds checking built in. For older languages, such as C, we can try to retrofit the C compiler to achieve this bounds checking. As we now discuss, implementing bounds checking in C is challenging.

One way to implement bounds checking in C is a technique called *fat pointers*. When using fat pointers, the compiler changes the representation of a pointer to include not only an address, but also the *base* and the *limit* of the buffer the pointer points to. This base and limit are initialized on an allocation, and the compiler inserts bounds checks on each pointer dereference to guarantee that the dereferenced value is within the array bounds that the base and limit specify. Pointer arithmetic preserves the base and limit but modifies the pointer itself as before, allowing the pointer to possibly go out of bounds.

For example, if a programmer allocates an array of 128 bytes using `void* ptr = malloc(128)`, the compiler will associate values `base = ptr` and `limit = ptr+128` with the pointer `ptr`. If we then assign `ptr3 = ptr + 3`, then the new pointer `ptr3` will have the same base and limit as `ptr`:



When the program deferences `ptr3`, the compiler will insert checks to ensure that `base` $\leq$ `ptr3` $<$ `limit` and crash the program otherwise.

One limitation of fat pointers is that they only check overflowing an *allocated region of memory*—not overflows *within* a region of memory. For example, if we use `malloc` to allocate a C `struct`, overflowing the `buf` member of the struct could allow the attacker to overwrite the values of other elements of the struct:

```
struct {
  int x;
  char buf[16];
  void (*f)();
}
```

If there are function pointers in the `struct`, the result of this within-region overflow could be as bad as overflowing the return address.

Unlike a nonexecutable stack, stack canaries, and ASLR, fat pointers are not widely used. This is largely because the modified "fat" pointers can break the functionality of existing C code. In particular, a fat pointer on a 64-bit architecture will typically take more than 64 bits to represent. If the programmer cast a pointer to an `int` and back again, the behavior of the program could change when using fat pointers versus when using unmodified 64-bit pointers. In addition, a C program may implicitly require that a pointer takes exactly 64 bits to represent (e.g., as a field within a `struct`); a compiler using fat pointers will break such programs.

1.5 Control-Flow Integrity (CFI)

The goal of techniques for *control-flow integrity* is to limit the set of addresses that a program will jump to when returning. In this way, a victim process may be able to detect when it is about to jump to a return address that an adversary modified with a buffer overflow. Implementing CFI requires compiler support—many modern compilers support some form of CFI.

To implement CFI, we need to add several checks on different kinds of jumps.

- For direct jumps (e.g,. "`call gets`", we know at compile time that these jumps are valid since there is nothing that the attacker can control. There is no need to implement any CFI checks on these jumps.
- For indirect jumps that use some variable in the jump target (e.g., function pointers, function returns), the compiler inserts checks to insure that the return address is a valid jump target. To do this, the compiler inserts into the program a data structure that maintains a set of all valid indirect-jump targets. Checking whether a jump point is valid takes a bit of extra computation—and thus imposes some runtime cost—it may tolerable in practice.

Often a compiler will implement this data structure using some sort of simple Bloom filter.

1.6 Unsolved program: Use-after-free bugs

The defenses in this section have made buffer overflows extremely challenging to exploit in modern code on modern systems. These defenses do very little to prevent against *use-after-free* bugs, in which a program `frees` allocated memory and then inadvertently reads or writes the `freed` pointer.

2 *Taint Tracking to Defend against Input-Sanitization Bugs*

We have discussed SQL injection and cross-site scripting, which allow an attacker to run code by adding special characters, such as "`"`" or "`>`," into their input.

Unlike buffer-overflow bugs, input-sanitization bugs arise not because a bug at one particular point in the program—they are more due to a systematic failure to consider certain types of inputs. As a result, the defenses are more systematic as well.

2.1 *Taint tracking in libraries*

A common approach to check for sanitization failures at runtime is called *taint tracking*. In taint tracking, some infrastructure in the program (e.g., a SQL library) marks any data coming from user input as *tainted*. At functions that perform escaping, the infrastructure removes the taint label. The infrastructure marks sensitive functions, such as the HTML renderer, are marked as *sinks*. Any time the program runs sink code, the taint-tracking infrastructure checks that the data headed into the sink 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 infrastructure will halt the program to avoid an exploit. See Fig. 1 for an example.

```
name = read_from_user()     # name is tainted
first,last = name.split()   # first,last are tained
# query is tainted
query = "SELECT␣*␣FROM␣USERS␣WHERE␣last␣name␣=␣'" + last "'"
query_database(query)       # will cause an exception

qesc = escape(query)        # qesc is not tainted
query_database(qesc)        # will succeed
```

Figure 1: A hypothetical Python example of how taint-tracking might prevent SQL injection

Many browsers implement taint tracking 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 explicity 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.

2.2 *Taint tracking in language runtimes*

Some programming languages associate a *taint bit* with every string in the program. The application developer can set or clear the taint bit manually. In addition, the language runtime will automatically set the taint bit on strings that certain functions (e.g., those that read from user input) return. An application developer then can implement some taint-checking policy to systematically prevent against escaping bugs. For example, if there is one function that makes a SQL query to the database, the application could check the taint bit is cleared on any query string that a developer passes to this function.

2.3 *Taint tracking in operating systems*

Operating systems also use taint tracking, often to flag suspicious files—typically those that the user downloaded from the Internet. MacOS, for example, will set a taint bit on any executable that the user downloaded from the Internet. The OS will maintain this taint bit when the user copies or moves a tainted file. If the user ever tries to execute a tainted file, the OS will raise a warning to the user before executing it.

References

Bittau, Andrea et al. “Hacking blind”. In: *IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 227–242.