CS 161 Computer Security
Spring 2010 Paxson/Wagner
Answers 1

January 26, 2010

1. Compiler defense

There is a technique for defending against stack smashing attacks called "canaries." Similar to a "canary in a coal mine," a canary in this context is a variable that will indicate whether or not a stack smashing attack has occurred. This technique is used by a compiler, not by a programmer. The idea is that the compiler inserts some additional code inside of each function to detect when a stack smashing attack has occurred, and then, during runtime, this code halts execution if an attack has been detected.

How might this work?

Hint: Think about which parts of memory get overwritten during a buffer overflow. Drawing a diagram of the stack may help.

Answer: The compiler sets a global number randomly before each run. This number is the magic "canary" number. Take this function:

```
void vuln() {
    char buf[n];
    gets(buf);
}
```

The compiler will take this function and generate:

```
static const int MAGIC = 7; // This number is randomly set before each run.

void vuln() {
    char buf[n];
    int canary = MAGIC;
    gets(buf);
    if (canary != MAGIC)
        HALT();
}
```

StackGuard is an example of a real program that uses this exact defense.

2. OS defense

Professor Wagner mentioned during lecture that for a malicious code injection attack, you can get the address

of the start of your malicious code by running the debugger. He also said that this tends to stay the same between different runs of the program.

What could the OS do about this to make it harder for an attacker to accurately insert the address of the start of his malicious code?

What types of attacks will this not defend against?

Answer: The solution is something called "Address Space Layout Randomization," or ASLR. The OS gives a big chunk of virtual memory to the program. However, instead of deterministically allocating the layout, it places things randomly in the address space. For instance, in might decide to start stack frames from somewhere other than the highest memory address.

This only addresses malicious code injection. However it does not defend against local data manipulation (e.g. changing an authentication flag).

This defense completely depends on how much randomization you have, however. Remember, the attackers can keep trying. If they try enough, and your randomization is not in a wide enough range, eventually they will still win.

3. Another OS defense

We learned that in a malicious code injection attack, attackers often store their malicious code inside the buffer that they overflow.

What could the OS do to stop this code from being executed? What types of code would break with this defense in place?

Hint: Is there some property of these memory pages that indicates that code contained in them should not be executed?

Answer: Modern operating systems provide a feature called `W^X` ("Write xor execute") pages. Effectively, a page in memory can never be both writable and executable. The problem is that this disallows self modifying code.

4. Direction of stack growth

When the stack grows down (i.e. from higher memory addresses to lower memory addresses), a buffer overflow of a stack-allocated local variable can overwrite the return address for that stack frame.

What if the stack grows up (i.e. from 0 to higher memory addresses) instead? Are we safe from stack smashing attacks? Why or why not?

Hint: Consider all stack frames in this vulnerable code:

```
void vulnerable() {
    char buf[80];
    gets(buf);
}
```

Answer: This does not stop overwriting the return address of functions called afterwards, not to mention local data.

5. **Memory-safe languages**
   Java doesn't allow a programmer to express a buffer overflow or other memory- safety vulnerability.

   What is the cost of this?

   Should the next version of C++ add memory safety? Why or why not?

   If the next version did add memory safety, could we all (finally) stop talking about buffer overflows? Why or why not?

   Answer: The cost is in performance because it is necessary to check bounds on each buffer access. C++ probably will not add it because people use the language to focus on performance.

   It also does not solve the problem because code has been written for dozens of years, and source code may not even be available anymore.

6. **Arc injection**
   Imagine that you are trying to exploit a buffer overflow, but you notice that none of the code you are injecting will execute for some reason. How frustrating! You still really want to run some malicious code, so what could you try instead?

   Hint: In a stack smashing attack, you can overwrite the return address with any address of your choosing.

   Answer: You might overwrite the address with a function in libc, for example. You can call fork() with a program of your choosing. This means, however, that you have to setup the stack appropriately.

7. **Vulnerable code**
   Is the following code safe? If so, why? If not, describe how to attack it.

```
void maybeSafe(size_t len, char *data) {
    char *buf = malloc(len+2);
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len+1] = '\0';
}
```

   Answer: It is not safe. There is a heap buffer overrun if `len` $>$ `UINT_MAX - 2`.