Paxson Spring 2017 — CS 161 Computer Security — Discussion 2

Question 1 ***Software Vulnerabilities*** **(15 min)**

For the following code, assume an attacker can control the value of **basket** passed into **eval_basket**. The value of **n** is constrained to correctly reflect the number of elements in **basket**.

The code includes several security vulnerabilities. **Circle *three* such vulnerabilities** in the code and **briefly explain** each of the three.

```
1  struct food {
2    char name[1024];
3    int calories;
4  };
5
6  /* Evaluate a shopping basket with at most 32 food items.
7     Returns the number of low-calorie items, or -1 on a problem. */
8  int eval_basket(struct food basket[], size_t n) {
9    struct food good[32];
10   char bad[1024], cmd[1024];
11   int i, total = 0, ngood = 0, size_bad = 0;
12
13   if (n > 32) return -1;
14
15   for (i = 0; i <= n; ++i) {
16     if (basket[i].calories < 100)
17       good[ngood++] = basket[i];
18     else if (basket[i].calories > 500) {
19       size_t len = strlen(basket[i].name);
20       snprintf(bad + size_bad, len, "%s ", basket[i].name);
21       size_bad += len;
22     }
23
24     total += basket[i].calories;
25   }
26
27   if (total > 2500) {
28     const char *fmt = "health-factor --calories %d --bad-items %s";
29     fprintf(stderr, "lots of calories!");
30     snprintf(cmd, sizeof cmd, fmt, total, bad);
31     system(cmd);
32   }
33
34   return ngood;
35 }
```

Reminders:

- **strlen** calculates the length of a string, not including the terminating '\0' character.
- **snprintf(buf, len, fmt, ...)** works like **printf**, but instead writes to **buf**, and won't write more than **len - 1** characters. It terminates the characters written with a '\0'.
- **system** runs the shell command given by its first argument.

Solution: There are significant vulnerabilities at lines **15/17**, **20**, and **31**.

Line **15** has a fencepost error: the conditional test should be $i < n$ rather than $i <= n$. The test at line **13** assures that **n** doesn't exceed 32, but if it's equal to 32, and if all of the items in **basket** are "good", then the assignment at line **17** will write past the end of **good**, representing a buffer overflow vulnerability.

At line **20**, there's an error in that the length passed to **snprintf** is *supposed* to be available space in the buffer (which would be **sizeof bad - size_bad**), but instead it's the length of the string being copied (along with a blank) into the buffer. Therefore by supplying large names for items in **basket**, the attacker can write past the end of **bad** at this point, again representing a buffer overflow vulnerability.

At line **31**, a shell command is run based on the contents of **cmd**, which in turn includes values from **bad**, which in turn is derived from input provided by the attacker. That input could include shell command characters such as pipes ('|') or command separators (';'), facilitating *command injection*.

Some more minor issues concern the **name** strings in **basket** possibly not being correctly terminated with $'\backslash 0's$, which could lead to reading of memory outside of **basket** at line **19** or line **20**.

Note that there are no issues with format string vulnerabilities at any of lines **20**, **29**, or **30**. For each of those, the format itself does not include any elements under the control of the attacker.

Question 2 ***Buffer Overflow Mitigations*** **(20 min)**

Buffer overflow mitigations generally fall into two categories: (1) eliminating the cause and (2) alleviating the damage. This question is about techniques in the second category.

Several requirements must be met for a buffer overflow to succeed. Each requirement listed below can be combated with a different countermeasure. With each mitigation you discuss, think about *where* it can be implemented—common targets include the compiler and the operating system (OS). Also discuss limitations, pitfalls, and costs of each mitigation.

(a) The attacker needs to overwrite the return address on the stack to change the control flow. Is it possible to prevent this from happening or at least detect when it occurs?

(b) The overwritten return address must point to a valid instruction sequence. The attacker often places the malicious code to execute in the vulnerable buffer. However, the buffer address must be known to set up the jump target correctly. One way to find out this address is to observe the program in a debugger. This works because the address tends to be the same across multiple program runs. What could be done to make it harder to accurately find out the address of the start of the malicious code?

(c) Attackers often store their malicious code inside the same buffer that they overflow. What mechanism could prevent the execution of the malicious code? What type of code would break with this defense in place?

Solution:

(a) **Stack Canaries.** A *canary* or *canary word* is a known value placed between the local variables and control data on the stack. Before reading the return address, code inserted by the compiler checks the canary against the known value. Since a successful buffer overflows needs to overwrite the canary before reaching the return address, and the attacker cannot predict the canary value, the canary validation will fail and stop execution prior to the jump.

As an example, consider the following function.

```
void vuln()
{
    char buf[32];
    gets(buf);
}
```

The compiler will take this function and generate:

```
/* This number is randomly set before each run. */
int MAGIC = rand();

void vuln()
{
    int canary = MAGIC;
    char buf[32];
    gets(buf);
    if (canary != MAGIC)
        HALT();
}
```

Limitations.

- Canaries only protect against stack smashing attacks, not against heap overflows or format string vulnerabilities.
- Local variables, such as function pointers and authentication flags, can still be overwritten.
- No protection against buffer *underflows*. This can be problematic in combination with the previous point.
- If the attack occurs before the end of the function, the canary validation does not even take place. This happens for example when an exception handler on the stack gets invoked before the function returns.
- A canary generated from a low-entropy pool can be predictable. In 2011 research showed that the Windows canary implementation only relied on 1 bit of entropy.

Cost. The canary has to be validated on each function return. The performance overhead is only a few percent since a canary is only needed in functions with local arrays. To determine whether to use the canary, Windows additionally applies heuristics (which unfortunately can also be subverted.)

(b) **Address Randomization.** When the OS loader puts an executable into memory, it maps the different sections (text, data/BSS, heap, stack) to fixed memory locations. In the mitigation technique called *address space layout randomization* (ASLR), rather than deterministically allocating the process layout, the OS randomizes the starting base of each section. This randomization makes it more difficult for an attacker to predict the addresses of jump targets. For instance, the OS might decide to start stack frames from somewhere other than the highest memory address.

Limitations.

- Entropy reduction attacks can significantly lower the efficacy of ASLR. For example, reducing factors are page alignment requirements (stack: 16 bytes, heap: 4096 bytes).
- Address space information disclosure techniques can force applications to leak known addresses (e.g., DLL addresses).
- Revealing addresses via brute-forcing can also be an effective technique when an application does not terminate, e.g., when a block that catches exceptions exists.
- Techniques known as *heap spraying* and *JIT spraying* allow an attacker to inject code at predictable locations.
- Like the canary defense, ASLR also does not defend against local data manipulation.
- Not all applications work properly with ASLR. In Windows, some opt out via the `/DYNAMICBAS` linker flag.

Cost. The overhead incurred by ASLR is negligible.

(c) **Executable Space Protection.** Modern CPUs include a feature to mark certain memory regions non-executable. AMD calls this feature the NX (**n**o e**x**ecute) bit and Intel the XD (e**x**ecute **d**isable) bit. The idea is to combat buffer overflows where the attacker injects their own code.

PaX pioneered this technique in 2000 with per-page non-executable page support, protecting binary data, the heap, and the stack. OpenBSD implented a form of this called W⊕X (write x-or execute) in 2003. Since Service Pack 2 in 2004, Windows features *data execution prevention* (DEP), an executable space protection mechanism that uses the NX or XD bit to mark pages, which are intended to only contain data, as non-executable.

Limitations.

- An attacker does not have to inject their own code. It is also possible to leverage existing instruction sequences in memory and jump to them. See part 3 for details.
- The defense mechanism disallows execution of code generated at runtime, such as during JIT compilation or self-modifying code (SMC).
- If code is loaded at predictable addresses, it is possible to turn non-executable into executable code, e.g., via system functions like `VirtualAlloc` or `VirtualProtect` on Windows.

Cost. There is no measurable overhead due to the hardware support of modern CPUs.

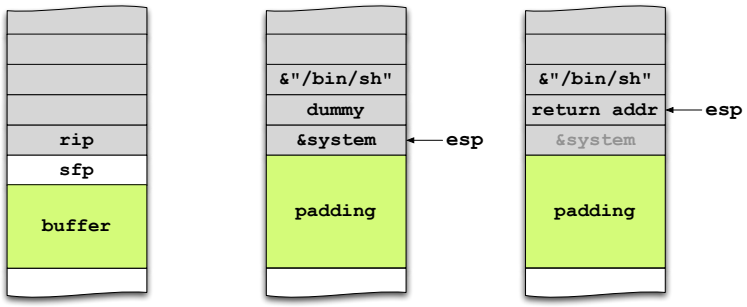
Question 3 ***Arc Injection*** **(15 min)**

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.

Solution: Rather than injection code, the main idea of *arc injection* is to inject data. It is a powerful technique to that bypasses numerous protection mechanisms, in particular executable space protection (2a:exec). By injecting malicious data that existing instructions later operates on, an attacker can still manipulate the execution path.

For example, an attacker can overwrite the return address with a function in `libc`, such as `system(const char* cmd)` whose single argument `cmd` is the new program to spawn. The attacker also has to setup the arguments (i.e., the data) appropriately. Recall that function arguments are pushed in reverse order on the stack before pushing the return address. Consider the example below, where an attacker overwrites the return address with the address of system (denoted by `&system`) to spawn a shell.



The first figure on the left is the stack layout before the attack. The second figure in the middle represents the state after having overflowed the buffer. Here, the return address is overwritten with `&system`. The value above is the location of the return address, from the perspective of `system`'s stack frame. But since the attacker plans on spawning a shell that blocks to take evil commands (e.g., `rm -rf /`), this value will never be used — hence any dummy value will suffice. The argument to `system` is the address of attacker-supplied data, in this case a pointer to the string `/bin/sh`. Finally, the third figure displays the stack state after transferring control to `system`, which happened by popping `&system` into the program counter (and decrementing the stack pointer). At this point, the attacker can execute commands using the shell.

A more sophisticated version of arc injection is called *return-oriented programming* (ROP). It is based on the observations that the virtual memory space (which has the C library) offers many little code snippets, *gadgets*, that can be parsed as a valid sequence of instructions and end with a `ret` instruction. Recall that the `ret` instruction is equivalent to `popl %eip`, i.e., it writes the top of the stack into the program counter. The attacker does not even have to jump to the start of a function, any arbitrary location in the middle works as long as it terminates with a `ret`.

Shacham et al. showed that these small gadgets can be combined to perform arbitrary computation. In our above example, a basic combination of two gadgets would involve writing the starting address of the next gadget at the value of `dummy`. When the first gadget finishes, the next one is loaded by executing `ret`.

Setting up the stack is very tricky to get right manually, but the paper referenced above actually wrote a compiler to transform code from a language as expressive as C-like into mixture of gadgets to be pushed on the stack!

A final note: do not hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.