CS11 BINARY EXPLOITATION – LECTURE 2

Intro to Memory Corruption

What is Memory Corruption?

- Modifying a program's memory in some unexpected or unintended way.
- The term is a bit of a catch-all, almost all vulnerabilities you'll deal with for the rest of term will be memory corruption.

What is Memory Corruption?

- Really, really fun ☺

What is memory corruption useful for?

- Typically, we want to modify a program's control flow
- Programs take actions based on the contents of memory
  - CPU state stored on stack
  - Program state often stored in heap, or on stack too
- Corrupting memory allows for redirecting control flow!
- Almost all "exploits" will rely on some kind of memory corruption.

Terminology

- Typically, an invalid access by a program is called a **segfault**.
  - This is a “segmentation fault”, a relic from when program’s 32-bit addresses were split into two 16-bit words – the [XXXX]:[YYYY]
  - XXXX was the “segment”
  - A segmentation fault told the programmer they were accessing an invalid segment (one that didn’t exist).

Terminology

- I personally don't like the term segfault.
  - It's a relic that doesn't really make sense in the context of modern memory management.
  - Different kinds of faults can be trapped with different exception-handling vectors, but "segfault" conflates the different kinds into one…
- Feel free to use it, but I'll encourage the following terms:
  - **Data abort.**
    - This is the traditional segfault – a "mov" instruction (or similar) causes the CPU to try to access a non-existent address.
  - **Prefetch abort.**
    - When the **instruction pointer** is set to an invalid address. The CPU's pipeline will try to prefetch the instruction, and abort.
  - **Undefined instruction.**
    - When the CPU tries to decode an instruction, and fails.

Understanding memory corruption through case study

- For most of the rest of lecture, we’ll be looking at the following (contrived) program. Our goal is to find some way to get control of the instruction pointer.

```
int main(int argc, char **argv) {
    char my_number[16];
    int scratch[10];
    int index;
    int value;
    while (1) {
        printf(“Enter an index:\n”);
        index = atoi(gets(my_number));
        printf(“Old value: %d\n”, scratch[index]);
        printf(“Enter a new value:\n”);
        value = atoi(gets(my_number));
        scratch[index] = value;
        printf(“New value: %d\n”, scratch[index]);
        printf(“Stop?”);
        if (atoi(gets(my_number)) != 0) {
            break;
        }
    }
    return 0;
}
```

Case study – out-of-bounds indexing

```
int main(int argc, char **argv) {
    char my_number[16];
    int scratch[10];
    int index;
    int value;
    while (1) {
        printf(“Enter an index:\n”);
        index = atoi(gets(my_number));
        printf(“Old value: %d\n”, scratch[index]);
        printf(“Enter a new value:\n”);
        value = atoi(gets(my_number));
        scratch[index] = value; // Both index and value are controlled.
        printf(“New value: %d\n”, scratch[index]);
        printf(“Stop?”);
        if (atoi(gets(my_number)) != 0) {
            break;
        }
    }
    return 0;
}
```

Case study – out-of-bounds indexing

- What happens when an array is read from a negative index? What about an index that's too big?
  - Sometimes, you'll get a segfault.
  - But a lot of the time, you won't!
- Low-level languages assume you know what you're doing.
- Array accesses are translated (typically), as *(array_start + index * sizeof(array_element)).
- arr[-1] will access the value in memory before the start of the array, and arr[arr_len] will access the memory just past the end!

Case study – out of bounds indexing

- Our function lets us do the following things with a local variable array:
  - Read arbitrary indexes
    - printf(“Old value: %d\n”, scratch[index]);
  - Write arbitrary indexes
    - scratch[index] = value;
- Gives us the ability to read and write from **arbitrary memory**.
  - In a +/- 2^32 range, anyway.
- What can we do with this?
  - We’ll need to understand how memory is laid out…

Case study – stack frame

- From low to high (remember from CS24), the stack will look a little like this:

- [ ] <- ESP
- [callee-saved registers]
- [local variables]
- [base pointer] <- EBP
- [return address]
- [arguments]

Case study – out-of-bounds indexing

- Function had the following local variables:

```
char my_number[16];
int scratch[10];
int index;
int value;
```

- On the stack, they'll be laid out in order from low-to-high.
  - scratch[-1] reads the last four bytes of my_number.
    - Lower than that will read callee save registers, if any (compiler specific)
  - scratch[10] reads index;
  - scratch[11] reads value;
- What about scratch[12]?

Case study – out-of-bounds indexing

- [local variables]
- [base pointer] <- EBP
- [return address]

- scratch[12] will read the low part of the base pointer!
  - Assuming 64-bit pointers, and a "simple" compiler.
  - All of this can be adjusted as needed.
- More interestingly, scratch[14] and scratch[15] will read and write the **return address**.

Review – Return Addresses

- When a function gets called, the program needs to know where to return to afterwards.
- The address to jump to after the function finishes is **pushed onto the stack**.
- When the function finishes, the **ret** instruction will be used to **pop the return address directly into the instruction pointer**!
- We want control of the instruction pointer.
- This is great!

Case study

- We can read the contents of the stack, and modify the return address of main to be whatever we want.
  - It seems like we should be able to do quite a lot with this! ☺
- However, before we move on to actually exploiting this, let’s talk about another way of doing (basically) the same thing: **buffer overflows**.

Buffer Overflows

- What is a buffer overflow?
  - When some code writes values past the end of a buffer!
- There’s a (trivial) one in the example program..

Buffer Overflows

```
int main(int argc, char **argv) {
    char my_number[16];
    int scratch[10];
    int index;
    int value;
    while (1) {
        printf("Enter an index:\n");
        index = atoi(gets(my_number));
        printf("Old value: %d\n", scratch[index]);
        printf("Enter a new value:\n");
        value = atoi(gets(my_number));
        scratch[index] = value;
        printf("New value: %d\n", scratch[index]);
        printf("Stop?");
        if (atoi(gets(my_number)) != 0) {
            break;
        }
    }
    return 0;
}
```

Buffer Overflows

- Gets is a very unsafe way of getting input.
- It reads from stdin and writes to a provided buffer **until stdin runs out of characters**!
- my_number has a fixed size (16)
- If there are more than 16 characters to read from stdin, gets will happily write them past the end of my_number!

Buffer Overflows

- We can take advantage of this just like we can take advantage of out-of-bounds idnexing.
- If we provide more than 16 characters to stdin, we can overwrite scratch, then index, then value, then the base pointer, and then a **return address**.
- Buffer overflows sound complicated, but they're really very simple!
  - They're just a special, limited case of out-of-bounds indexing
  - Limited to writing, not reading
  - Usually only for large positive values and not negative ones.
- End goal (overwriting a return address) is **exactly the same!**
- This kind of buffer overflow on the stack is also called a **stack overflow**.
  - That website you all use is named after this type of vulnerability.

Modifying Return Addresses

- So we can read/write to a return address.
  - We could use the buffer overflow, or we could use the oob r/w.
  - Either is fine, but the oob r/w is probably simpler ☺
- We can make the program jump to any address we want!
- How do we actually do something interesting with this?
  - For now, let’s take the path of least resistance.
- Solution: look for pre-existing code!

Modifying Return Addresses

- Ideally, we'd like to have our program launch a shell, so that we can explore the computer the program is running on.
- Sometimes you'll have the luxury of having functions that do exactly what you want already in the main program.
  - In the first assignment, for example, there will be functions in memory that will do exactly what you want.
- For now, we'll assume this is the case.
- Makes things simple: just set the return address to the function we want to call!
- Next week: We'll talk all about the kinds of fun stuff you can do if you **don't** have a perfect function you can jump to.

This week's set

- Two capture-the-flag type programs.
  - An out-of-bounds indexing error
  - A buffer overflow
- Both should be pretty simple!
  - Your job will be to figure out what kinds of input will allow you to recover the flag (password).
- You'll be able to play with the programs locally (to debug), but to get the flag you'll need to run your solution against an online version.
  - This will be served from an aws vm via netcat. ip/port pending.

Review – Providing non-standard input to programs

- A lot of this term, you’ll be wanting to make programs take in non-ascii input.
  - This can be harder to do on your end than it sounds!
- I’ll go over two ways to do this, so that you don’t have to struggle against it on your own.

Non-standard input: the easy way

- Unix pipes (|) allow you to set the standard input of one program to the standard output of another.
  - Set the input of the program you're targeting to the output of a different program that generates the input you want!
- Bash commands are your friend:
  - echo –e '\xaa\xbb\xcc\xdd' | some_program
  - python –c 'print "\xaa\xbb\xcc\xdd"' | some_program
  - Many other ways (printf, compiled programs, etc…)
- Can also put use your command as an argument:
  - some_program `echo –e '\xaa\xbb\xcc\xdd'`
  - some_program $(echo –e '\xaa\xbb\xcc\xdd')

Non-standard input: fancier tools

- There are lots of fancy toolkits for this kind of thing on the internet.
- In particular, I would recommend **pwntools**.
  - Some terminology I've left out until now: to "**pwn**" a program is slang for exploiting a vulnerability in it (to get code execution, etc).
- pwntools.com
- Pretty easy-to-use, and very feature-rich!
- You may want to play around with it and use it on the later sets ☺

Next week

- All about modifying return addresses.
  - With a brief interlude on buffer overflows that affect the *heap*, and not the stack.
- We'll talk about shellcode (writing your own code into memory), and about what you can do even when shellcode isn't an option.