CS681: Memory Corruption Vulnerabilities

Polytechnic University
Stanislav Nurilov
December 1, 2005
Tonight I will try to give a crash course in at least 3 different computer topics all in 2 hours (hopefully less)

Those who don’t follow what I’m saying, please ask.

However, I expect most/all of you to be familiar with everything I say.

Some of you have already exploited Memory Corruption Vulnerabilities. (I hope I don’t bore you.)
1: How does computer work.

- There is a processor that interfaces with other devices.
- Processors execute instructions.
  - A processor knows about some number of instructions and can execute from that set.
  - Programmers use the basic instructions provided to create more complex functions.
    - In the beginning people wrote binary
    - Then people wrote in assembly and converted to binary
    - Then people wrote in C (I skipped a couple of languages there), translated to assembly and then converted to binary.
    - As time goes on more and more people forget how to program in assembly (cause they don’t have to)
2: Where to get instructions and data?

- Instructions are loaded into memory. Often this is the same memory that data is loaded into as well.

- Every process on modern operating systems using x86 architecture thinks that it has 4GB of addressable memory space. (RAM)
  - OS makes sure that this fantasy comes true for processes. (virtual memory management)
3: How do computers look at information?

- What is 10010000?
  - 0x90 (hex)
  - 0220 (octal)
  - 10,010,000
  - 144 (decimal)
  - “É” (character)
  - x86 NOP instruction

- Rule #1: computer systems can view binary data as anything that they would like. (data as instruction) It depends on the context.
4: So how does it work?

- Don’t worry about OS.
- Assume that the process is running on entire machine, that’s what it thinks anyway. OS makes this fantasy come true.
- Read instructions from memory, one by one and Process them. (add, sub, mul, div, etc.)
- Jumps/Branches are hard.
5: Is it this easy?

- No. Anybody who has programmed knows that you have jumps and loops and conditions in code. There are also functions.
- How can a processor keep track of which functions it calls and which functions it comes back to.
- Use tricks like stack.
6: What is a stack?

- This is just a memory location somewhere in the 4GB space.
- When functions are called, the address of the instruction that the processor was at is stored on the stack.
- When function is over, Retrieve the saved pointer to next instruction from the stack.
Also store other useful data on the stack
- Function variables
- Function arguments

Each function gets “activation record” on stack
- What is an activation record? Just some memory location in the 4GB space.
- What does it do? It just says where the variables belonging to the function are on the stack.
8: Is that it?

- There is also a heap.
  - It is just a memory location in the 4GB space. It is used for dynamic allocation of memory.
  - Sometimes a programmer doesn’t know how much space he will need. So he can allocate it during run time.

- There is also a Data Section
  - It is just a memory location in the 4GB space.
  - It has the values of all global variables.

- There is more…
Does anybody see any problems?

- Everything is in the same 4GB space. So?
- Data can be treated as instructions and vice-versa. So?
- What if we can corrupt some memory location?
  - Nothing (No biggie)
  - Denial of Service
  - Change control flow of a program
  - Change functionality of a program
How do we do this?

- Architecture has to allow it.
- Programmer has to allow it. (Not on purpose.)
  - Danger of high level abstraction. People don’t know what happens under the hood.
- Attacker has to be able to get to it.
- **Unchecked user input accounts for every exploit out there.**
How do we prevent it

- Change architecture to be more secure, less forgiving.
- Educate Programmers about mistakes.
- Block attackers from getting to it.
Common Memory Corruption Techniques

- Static Buffer Overflows (Stack)
- Heap based Buffer Overflows (Heap)
- Double Free() (heap based)
- Memory Initialization Corruption (Stack, Heap)
- String Format attacks (Stack/Heap)
- Off by one errors
- Unchecked index, array access
- Others (not easily classifiable)
Common Elements of all Memory Corruption Vulnerabilities

- Have write access to a 4GB memory location where something important is stored.
  - Something that controls program flow
- Be able to execute something anywhere in 4GB memory location.
- More access to memory = better ability to exploit
What is a buffer overflow?
- A buffer refers to some space allocated by a computer program in memory. (Some memory location in 4GB space)
- This buffer space can be an array declared in a C program as follows:
  ```c
  char buffer[256]; //this allocates 256 bytes of memory.
  ```
- An overflow occurs when data in the buffer is accessed that is not allocated. In a C program this can happen by trying to assign a value to a buffer location as follows:
  ```c
  buffer[261] = 'S';
  ```
- Here a buffer overflow occurs because a location is accessed that has not been allocated.
Where do buffer overflows usually occur?

- Buffer overflows usually occur in code segments where programmers are not careful about checking or enforcing the length of buffers that are passed in for input.

- So if a user is allowed to supply arbitrarily long input and the program copies this input into a statically allocated memory location then a buffer overflow will occur.
Why can this be attacked?

- Buffer overflows are annoying.
- An unintended memory location is overwritten with an unintended value.
  - When undetected, it results in a memory location being corrupted with some value.
  - Sometimes it won’t have any side effects
  - Sometimes it might cause program crashes.
  - However, if a meaningful location can be overwritten with a meaningful value, then we’re in business.
- Indeed it is the case that meaningful locations exist on a stack and these locations are available to be overwritten when a buffer overflow occurs.
- A stack is used to store arguments to functions and other information such as the addresses of instructions that will be executed after functions return. Overwriting these values on the stack can allow an attacker to gain access to program execution.
int function(char * a) {
    char buff[256];
    if (a == NULL) return -1;
    strcpy(buff, a); return 1;
}

int main(int argc, char** argv) {
    func(argv[1]);
    return (0);
}
*In this picture and the ones on the following slides, the stack grows down, as is the convention. Lower addresses are on the bottom of the picture and higher address at the top of the picture.
Before string copy: Conceptual
%esp
argv[1]

address of buff

shellcode

Func tries to return
How do we know this address??

- If you took this class last semester
  - I would tell you to guess.
- Now, I’ll tell you that I have a special presentation just for that.