Exploiting Static Buffer Overflows

Presented by Stanislav Nurilov
September 14, 2005
Polytechnic University - SFS
Overview

- Static buffer overflow basics
- Bouncing off static libraries into shellcode
- Other approaches
- Explanation of demo, tools, and techniques
- Scenario, Testing Methodology, and Demo
- Questions
Exploiting Stack Based Buffer Overflow Vulnerabilities

What is a buffer overflow?

A buffer refers to some space allocated by a computer program in memory.

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.

Example

```c
int func(char * a){
    char buff[256];
    if(a==NULL) return;
    strcpy(buff, a);
}
```

```c
int main(int argc, char** argv){
    func(argv[1]);
    return(0);
}
```

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?

It is clear that buffer overflows are an annoyance to the normal execution of a program. When a buffer overflow happens, 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, but 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.

The x86 architecture uses a stack based process execution model. In this model, a special construct (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.

Reference: Aleph One, Smashing the Stack For Fun And Profit.
Objective

- Smashing the Stack for Fun and Profit introduced many of us to the exact inner workings of buffer overflow vulnerabilities and exploits.
- The technique shown by Aleph1 relied on making some guesses and assumptions regarding the address of the shellcode on a stack-based buffer.
- Using Aleph1’s method would not be effective against an exploit that had to work in the wild, where the address of the shellcode might change from one running instance to the other.
- Also on Windows boxes Stack addresses begin with 0x0012zzzz. Having the 0 byte is a problem.
- The technique presented allows for a more general exploit that will work across all instances of the running vulnerable program, specific to an operating system or operating system class.
Stack Contents, Normal Execution

Before function call

<table>
<thead>
<tr>
<th>%esp</th>
<th>int argc</th>
<th>char** argv</th>
</tr>
</thead>
</table>

During function call

<table>
<thead>
<tr>
<th>%esp</th>
<th>vuln_buff[0-4]</th>
<th>saved %ebp</th>
<th>saved return %eip</th>
<th>int argc</th>
<th>char** argv</th>
</tr>
</thead>
</table>

After function call

<table>
<thead>
<tr>
<th>%esp</th>
<th>int argc</th>
<th>char** argv</th>
</tr>
</thead>
</table>
What do we do

- Find out which shared libraries are used in the binary. (ldd command can be used on *nix)
  - This also provides the address offset.
- Find out where in the shared library the instruction `jmp *%esp` occurs.
- Add this to the address offset to calculate where in memory the `jmp *%esp` instruction will occur.
- Use this calculated address instead of guessing exactly where the stack buffer is located.
- After the address, overflow several more bytes with
  - `jmp -constant_offset` instruction.
  - The `constant_offset` is the number of overflowed bytes.
- Since the address and the instruction are always constant for each compilation of the program, then the exploit will work every time.
What happens

- Overflow the buffer correctly
- What the program thinks is the saved `%eip` is actually an address in the shared library.
  - Execute instruction at shared library address, which happens to be `jmp *%esp`
- At `%esp` execute the `jmp -constant_offset` instruction
  - This jumps to the beginning of the buffer where the shellcode is located.
- Begin executing shellcode
- Own the box
Stack Contents, Exploit

During function call before strcpy

%esp →

vuln_buff[0-4]
...
...
vuln_buff[252-255]
saved %ebp
saved return %esp
int argc
char** argv

During function call after strcpy

%esp →

vuln_buff[0-4]
...
...
vuln_buff[252-255]
Addr of jmp **%esp in shared library
jap -264
char** argv

After function call

%esp →

vuln_buff[0-4]
...
...
vuln_buff[252-255]
Addr of jmp **%esp in shared library
jap -264
char** argv
Variations

- Sometimes the address of the buffer or something really close is already stored in one of the registers.
  - Like `%eax`, `%ebx`, `%esi`, `%edi`, etc.

- Instead of jumping twice, you can search for a `jmp *%eax` instruction (or one of the above) and get to your shellcode directly.
Writing the Demo: v1.0

- **Objective:**
  - Create a simple demo, with a simple buffer overflow.
  - Pass vulnerable buffer through command line

- **Issues:**
  - Which compiler to use, that gives most freedom for compiling with options I want.
  - How to write shellcode for Windows
  - How to scan Windows loaded libraries for addresses
Writing the Exploit: v1.0

- Objective
  - Simple buffer overflow with technique presented.
  - Use system call to pass argument to windows command line.
  - Use debugger to make sure that exploit works.

- Issues
  - Windows command line sucks (or maybe it’s me, but I think it’s really Windows)
  - Weird characters in the command line arguments cause nothing to happen. (This is probably good, but not for demo purposes.)
  - How to attach debugger to exploited process to make sure of correctness?
Urghhhhhhhh
Writing the Demo: v2.0

- A more complex demo, but also more realistic.
  - Use Socket programming to create a server type application
  - Easy to debug, since always listening for connections and always running.
  - Easy to pass weird non-alpha numeric characters to application

- Issues:
  - Windows uses additional non standard conventions for socket programming
  - Windows doesn’t have fork, the more complex API is annoying
  - Have to rewrite entire demo
Writing the Exploit: v2.0

- More network programming to write exploit.
- Realize that windows shellcode is large.
- Realize that windows shellcode uses lots of stack space, crashing into injected shellcode.
- Prove that technique indeed works for windows.
Ughhhhhhhhhhh 2
Writing the Demo: v3.0

- **Objective:**
  - Fix the stack collision.
  - Find another method to inject the shellcode.
  - Write another exploit.

- **Issues:**
  - Probably a bad idea to attack own box.
Writing the Exploit: v3.0

- Write two exploits
  - One using jmp *%esp
  - Another using call %eax
- And now the demo
Tool and Reference Summary

- Metasploit (www.metasploit.com)
- MSDN (msdn.microsoft.com)
- Beej’s Guide to Network Programming
- IA32 Assembly Guides (Intel®)
- Microsoft® Platform SDK
- Microsoft® cl compiler
- IDA Pro Interactive Debugger and Dissembler
- Flat Assembler
- 010 Hex Editor
Lessons Learned Summary

- Windows Network programming requires the use of WSAStartup() before any other network programming calls and WSAcleanup() to complement the startup procedure. This is needed to initialize the Ws2_32.dll
- Windows doesn’t have fork(). There is another API for creating processes and threads.
- Windows shellcode tends to get rather large.
- Windows shellcode calls many procedures, which tends to fill up large amounts of stack space quickly.
- It is best to move the %esp register after your shellcode to avoid stack collision with the shellcode.
- Use the /link <library_name> command line option in cl to link libraries with source code. Make sure to list source files before the /link option.
- Not sure how to pass complex bytes (non-alphanumeric and white space) to the windows command line.
- Not sure how to debug a process not already loaded into memory, other than using IDA.
Scenario: Part 1

- Uber hacker Stan wants to attack Financial Aid Department, for the wrongs committed against him.
- He nmaps all ip address belonging to FAD
- He discovers that the 1337 port is open on one address.
- It is a well known fact that port 1337 runs the vulnerable.exe server which is freely available online.
Scenario: Part 2

- Stan decompiles vulnerable.exe and finds a very easy buffer overflow in the second function he looks at.
- Stan writes an exploit to reverse connect a shell to an ip address and port on his computer.
- Stan runs the exploit and gains access to FAD computer.
Scenario: Disclaimer

- Never hack Financial Aid Department
- It is a bad idea.
- You will get caught.
- This is a fictional scenario.
Questions