

HW 8

CS681 & CS392 Computer Security

Understanding and Experimenting with Memory Corruption Vulnerabilities

DUE 12/18/2005

1 Motivation

Memory corruption vulnerabilities to change program execution flow (e.g. Buffer overflows, Heap overflows, Format String attacks) are the predominant source of most of today's exploits and attacks. The other major source being data logic manipulation and insertion, such as many of the attacks against PHP and Perl (including SQL Injections). As security professionals we should have a good understanding of the risk posed by memory corruption vulnerabilities. The best way to understand the underlying problems with a memory corruption vulnerability is to experiment with one. This lab is designed to help you understand the current problems and solutions to memory corruption vulnerabilities. You will have to analyze code segments, answer critical questions, design an exploit, and then execute it against a vulnerable program. We will experiment with Static Buffer Overflows as discussed in class and use the return-to-libc technique to help us write an exploit.

2 References

- Static Buffer overflows
 - <http://mixter.void.ru/exploit.html>[∞]
 - "Smashing the Stack for Fun and Profit" Aleph One.
(<http://www.shmoo.com/phrack/Phrack49/p49-14>)[∞])
- Return-to-libc
 - Writing exploits using return to libc
(http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/arc_injection/arc_injection.html)[∞])
 - The advanced return-into-libc exploits (<http://www.phrack.org/phrack/58/p58-0x04>)[∞])
- Format String Attacks
 - <http://www.lava.net/~newsham/format-string-attacks.pdf>[∞]
 - Wikipedia, Format String Attack Entry
(http://en.wikipedia.org/wiki/Format_string_attack)[∞])
 - Advances in format string exploitation
(<http://www.phrack.org/show.php?p=59&a=7>)[∞])

- Shellcode and Opcode Database
 - The Metasploit Project. (<http://www.metasploit.com>)

3 Your Task

You will use buffer overflow and return-to-libc technique to exploit the following code:
myNameIs.c

```

1.  /* Originally written by Jason Larsen of INL. Modified slightly by Stanislav Nurilov for
    assignment.(11/29/05)*/
2.  #include<stdio.h>
3.  #include<stdlib.h>
4.  #include<strings.h>
5.
6.  char MyName[1024];
7.
8.  void PrintUsage(char* filename){
9.      printf("%s: <name> <index> <value>\n",filename);
10. }
11.
12. void SetName(char* name){
13.     char LocalName[1024];
14.     sprintf(LocalName, "Last Name is %s\n",name);
15.     printf(LocalName);
16.     snprintf(MyName, 1024, LocalName);
17. }
18.
19. void SetValue(int index, int value){
20.     int* pairs;
21.     pairs=calloc(1024, sizeof(int));
22.     printf("The old value is %i\n", pairs[index]);
23.     if (index > 1024){
24.         printf("Index out of range\n");
25.         return 0;
26.     }
27.     pairs[index]=value;
28.     printf("The new value is %i\n",pairs[index]);
29.     free(pairs);
30. }
31.
32. int main(int argc, char**argv){
33.     if (argc!=4){
34.         PrintUsage(argv[0]);
35.         return 0;
36.     }
37.     setuid(0);

```

```
38.  SetName(argv[1]);
39.  SetValue(atoi(argv[2]), atoi(argv[3]));
40.  return 0;
41. }
```

This program is known as "myNameIs." The name of the program is really silly and it doesn't do much, but it has some nice vulnerabilities in it. Before we go into how to exploit the code please install the myNameIs binary first. Follow these instructions:

1. Compile and build the code above to myNameIs. (gcc myNameIs.c -o myNameIs)
2. su to root, if you are not already root. (su -)
3. Copy the myNameIs binary to /usr/bin/. (cp myNameIs /usr/bin/)
4. Change myNameIs to be world executable. (chmod a+x /usr/bin/myNameIs)
5. Change myNameIs to have setuid. (chmod +s /usr/bin/myNameIs)
6. If everything worked out then, ls -la /usr/bin/myNameIs should output something like:
-rwsr-xr-x root root myNameIs

For the rest of the lab, you will be working under a non-root account, such as guest. Therefore make sure you have one of those available.

3.1 Analysis and Exploit Design

Answer the following questions to help you get a grasp of the situation:

1. The code has a buffer overflow and a format string attack vulnerability. Does the code have any other memory corruption errors? If so, explain what they are in detail.
2. This code runs as root (If you set it up as above). What can one possibly do with myNameIs if he/she were able to find a bug and exploit it?

3.1.1 Buffer Overflow Questions

Given the answers to the questions in the previous part and the references, answer these follow up questions:

1. What is shellcode?
2. What is an egg? (The TA made a mistake during the presentation regarding what an egg is. Please research it and find out what it really is.)
3. Identify the buffer that you will exploit with a buffer overflow vulnerability.
4. Identify where the input for the buffer comes from. Is the input validated?
5. Identify the length of the buffer.
6. Explain why this buffer is vulnerable to a buffer overflow attack.
7. Explain one way of how this vulnerability can be mitigated. (e.g. How would you change the code to prevent a buffer overflow exploit?)
8. Carefully look at the way the buffer gets a value assigned. Are there any caveats? Identify them if they exist and explain why they could be a problem.

9. Based on the buffer length, the caveat identified above, and your knowledge of the stack layout, identify how big the payload must be to overwrite the saved return pointer on the stack.
10. To use the return-to-libc attack, you will need to do the following.
 1. Pick a shared library, used by the *myNameIs* application. Find where it gets loaded in memory. (If you are using a *nix based system, you can use the `ldd` command. This will provide a list of shared libraries used by the *myNameIs* application and the location of where they get loaded in memory. If you are using Windows, then shared libraries are better known as DLLs. There are several ways to find out where a dll is loaded in an application.) Provide a list of shared libraries and loading address for the *myNameIs* application that you compiled.
 2. Find the offset of the 'jmp esp' instruction located in the shared library from the address of the shared library. The 'jmp esp' instruction is written as hex bytes 0xFF 0xE4. Find this offset for any of the libraries that you have listed above.
 3. Calculate and provide the actual address, by adding the base loading address and the offset. This is where the 'jmp esp' instruction will occur in the processor memory. This is the address that you will want to replace the stored return address with. Make sure that the address doesn't have any NULL bytes in it.
 4. Remember, that you will have to place a `jmp -<constant_offset>` instruction after the return address that you provided. This instruction will be 3 bytes long. At least how long does your payload have to be now that you know this in order to successfully jump into the shellcode.
 5. What is the constant offset that you need to jump, in order to get into the beginning of your shellcode. Make sure to pad your shellcode with a couple of NOP instructions (byte 0x90) so that if your offset is wrong by one or two bytes, you're covered.
 6. Use an x86 assembler to compile the `jmp -<constant_offset>` instruction, where the `<constant_offset>` is the value you got in the above question. This instruction should be the last 3 bytes of your payload.

3.1.2 Format String Questions

Answer the following questions also:

1. Why can we use a format string to attack *myNameIs*? Briefly explain where the vulnerability is and how you might go about attacking the application.
2. Is the attack string going to be on the stack?
3. Since the format string attack is on the stack, what useful information can we find out about the application? Demonstrate this by printing out the first 80 bytes of the stack.
4. Which of the bytes that you printed out look like they are important numbers. List them and explain why you think they are important.
5. If you could change any location on the stack (from the numbers that you have printed) which values would you change and to what? The what doesn't have to be a concrete value but can be a brief explanation of a possible value.

3.2 Exploit Development

You will now have to write the buffer overflow exploit.

3.2.1 Buffer Overflow Questions

1. Combine the relevant parts of section 3.1.1 to create the payload/egg that you will send to the myNameIs application.
2. Draw a memory map (similar to how it was done in class) of how the myNameIs process stack looks at the point of attack and overlay your exploit code on top of it.
3. Write the exploit based on your results from the previous sections.

3.3 Deployment

Remember that you should be compiling and running the attack programs under a non-root account. You can use the following shellcode (Linux systems) for your attack programs. You can also generate shellcode using the Metasploit Project Framework:

```
char shellcode[] = "\xeb\x1d\x5e\x29\xc0\x88\x46\x07\x89\x46\x0c\x89\x76\x08\xb0
                  \x0b\x87\xf3\x8d\x4b\x08\x8d\x53\x0c\xcd\x80
                  \x29\xc0\x40xcd\x80\xe8\xde\xff\xff\xff/bin/sh";
```

You can use the following format for the payload delivery mechanism.

1. `int main(){`
2. `char *egg;`
3. `..... FILL IN`
4. `//execute myNameIs and pass in the egg as parameter`
5. `execl("/usr/bin/myNameIs", "myNameIs", egg, 0);`
6. `return 0;`
7. `}`

3.3.1 Suggestions

The TA used the following code to check which addresses in memory have the JMP esp instruction (bytes 0xFFD0). Obviously this code is system dependent and needs to be modified. It worked for a program compiled under linux. The starting address was where the libc.so.6 file was loaded for the compiled process.

1. `/* Written by Stanislav Nurilov as an example. This code will work only on linux system that I was using.`
2. That is, the offset address need to be modified for your specific system.

```

3. This code will produce a Segmentation fault, but it should print out some addresses,
   before it does.*/
4. #include<strings.h>
5. #include<stdlib.h>
6. #include<stdio.h>
7.
8. #define START_LIBC_ADDR 0xb7ec4000
9. #define END_SCAN_ADDR 0xb8000000
10.
11. int main(int argc, char ** argv){
12.
13. unsigned int i;
14.
15. for(i=START_LIBC_ADDR; i<END_SCAN_ADDR; i++){
16.     if(memcmp((char*)i, "\xFF\xD0", 2) == 0){
17.         printf("Found jmp esp at %#x\n", i);
18.     }
19. }
20.
21. return 0;
22. }

```

4. Hand In

- Answers to above questions.
- Source code for the buffer overflow attack.
- Screenshots that show that you have succeeded. These screenshots should have a before and after picture showing the output from the "whoami" command (The first should show something that is not root, and the after should show root)

You should submit assignment through my.poly drop box no later than 12 AM midnight on the due date.

Use the following convention to name the file:

<First Name>_<Last Name>_<Lab#>.doc

Submit using the name of only one of the members in your group and put both of your names in the document.

REMEMBER IF YOU DO NOT USE THIS NAMING CONVENTION YOUR ASSIGNMENT WILL NOT BE GRADED AND YOU WILL NOT RECEIVE ANY CREDIT.

PLEASE SUBMIT HOMEWORK ON TIME.