Christopher LaJon Morgan
Oct 29, 2013

Project 8 Stack_Smashing CS 465

Classic Buffer Overflow

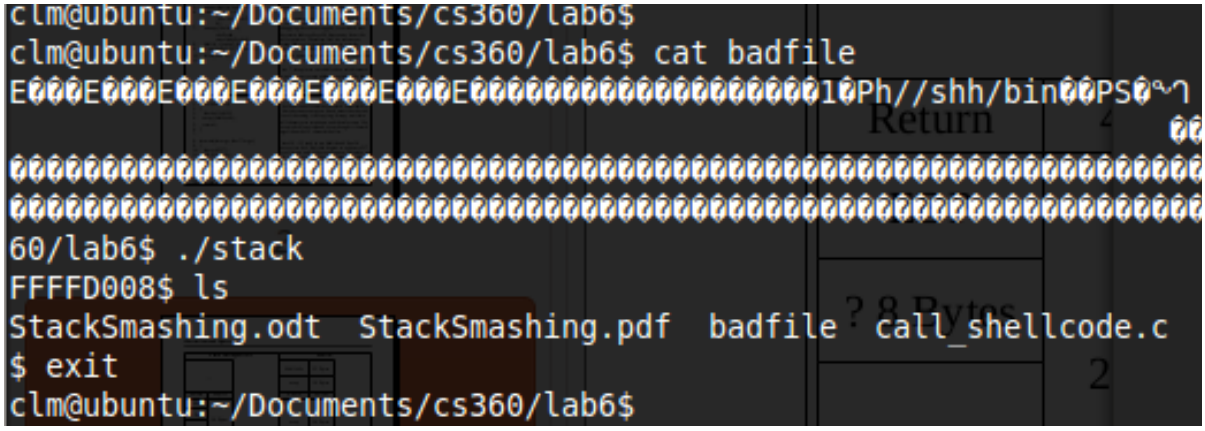
Vulnerable Code (Exe: ./stack):

```
int bof(char *str)
{ // removed Print $esp //
   char buffer[12];

   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);

   return 1;
}

int main(int argc, char **argv)
{
   char str[517];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 517, badfile);
   bof(str);

   printf("Returned Properly\n");
   return 1;
}
```

Generate Badfile:

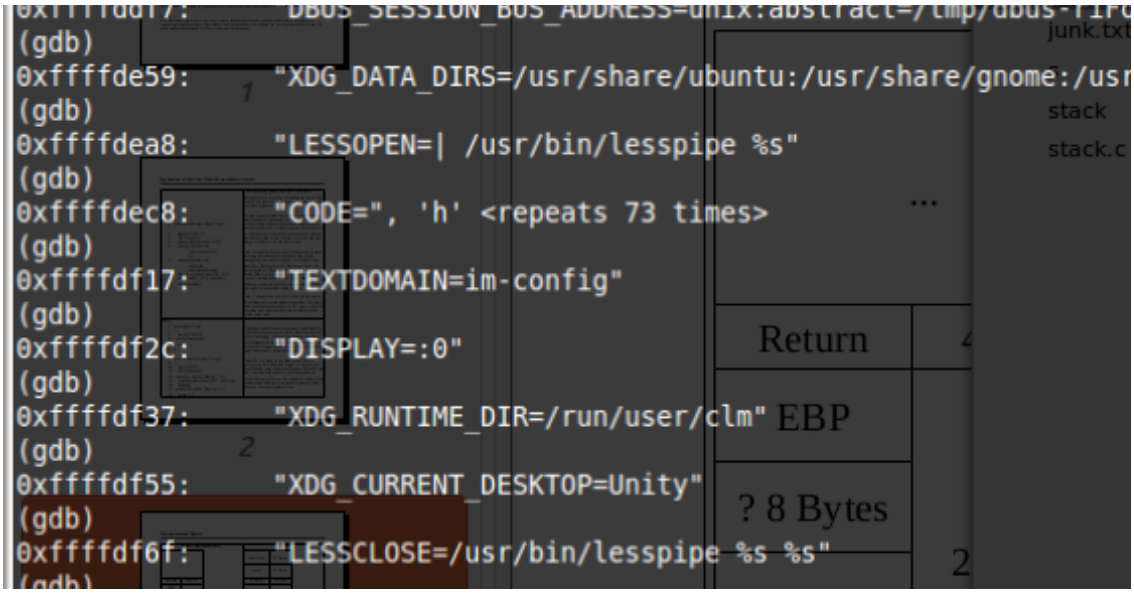
```
void main(int argc, char **argv)
{
   char buffer[517];
   FILE *badfile;
   int i = 0;

   /* Initialize buffer with 0x90 (NOP instruction) */
   memset(buffer, 0x90, 517);

   /* You need to fill the buffer with appropriate contents here */
   for (;i < 28; i = i + 4)
      memcpy(buffer + i, "\x45\xD0\xff\xff", 4);

   memcpy(buffer + 44, shellcode, sizeof(shellcode));

   /* Save the contents to the file "badfile" */  …
}
```

Smash:

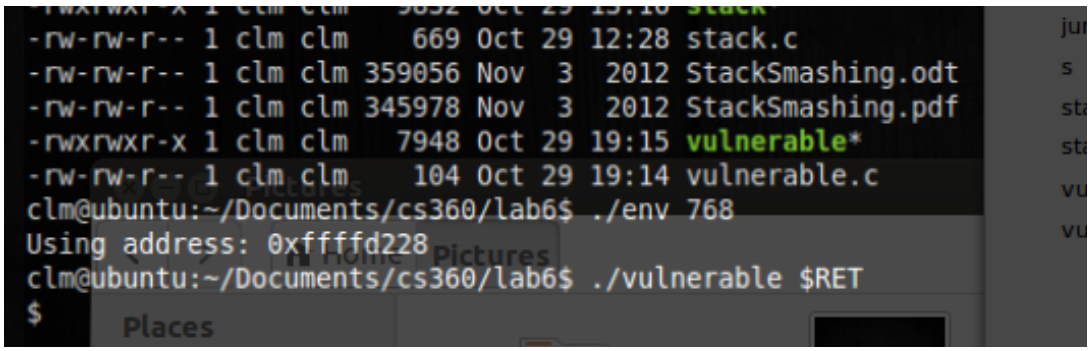


Environment Variable Exploit

Understand Inclusion of Env Variables (CODE):



Env Stack Smash Replication:



What was Learned

Buffer Overflow

- Very Implementation / Version Specific
  I had the opportunity to complete this project one year ago in 360. However, as I imported my old project it was completely broken. It ended up being nothing more than a framework in which to start working. I had to re-follow the entire process to update my code in order to perform this attack on my new linux box. In essence the stack information had completely changed. This required me to rewrite the placement of the return address and what the return address was.

- GDB
  In order to figure out where the return address should be placed and what it should be I had to relearn GDB. I used the disassemble command to figure out how far from the current stack pointer the return address was. I used the "x /s $esp" command to print the entire stack in order figure out what the injected return address should be.

- How a Buffer Overflow works
  Because my former implementation didn't work and the time spent with GDB I now have a greater understanding of the buffer overflow attack.

  Stack smashing is possible because of the way memory is organized and managed in a c program. In c program functions and data are stored on the stack. Program functions are stored on the stack in the sense that the stack is used to keep track of where execution should pick up after a function has completed. This is accomplished by pushing a return address onto the stack, which represents where the CPU should jump next to begin executing once the function is complete. Local data for functions are also stored on the stack below the return address. Because both user data and the return address (the code to execute next) are in the same space it allows the return address to be overwritten along with the data.

  The next important piece to the organization of memory is the way that it is laid out and written. Memory (the stack) is laid out from lower address to high, but the stack is written from high address to low. Therefore, when a function is called the return address is written above the local data. This also means that as local data is written, say like an array, it is written from the lower part of the stack up. This means that if an array is being written, from lower to high, it has the potential to run right over the return address if no precautions are taken to enforce array bounds.

  Stack smashing then utilizes this layout of memory to alter the programs flow. The goal is to overwrite the return address in the stack and point it to a new location that contains code that one has deviously written. It is accomplished by accessing local data, data that is below the return address, and then writing to it, without bounds checking, to the point of overwriting the return address. The return address is then redirected to an address in the stack that has executable code that one has written.

The code that I wrote follows this basic idea. Stack.c reads in my badfile file. It then calls a bad function in which attempts to copy my badfile into a very undersized buffer. This function does not check that its buffer is big enough to hold my file; therefore, my entire file is written from the start of buffer up through the stack. My file overlays the return address and much more because of this non-bounds checking function.

Therefore, Exploit.c very carefully crafts a badfile, such that it contains just enough data to place a calculated return address over top of the correct one. After the return address, my badfile contains machine executable byte code. This byte code then contains a series of no-op instructions and code to overlay the current program with a shell terminal. The no-op code makes the shell code easier to hit. This allows my badfile to open and run a terminal, when it should have just read in a file.

Now, how effective should this attack be today? It shouldn't be effective. Why, due to stack randomization and non-executable stack rules, stack smashing should be all but useless. Therefore it's important to note that in my experiments I modified the settings in my compiler and OS to turn off stack randomization and to allow full stack space code execution.

Environment Buffer Overflow

- Environment Variables are copied into the stack
  One completely new idea to me was the idea that environment variables are copied into a program's execution stack. This means that if one sets a new environment variable, I believe before a program is ran, the variable and its value, are copied directly into the program's stack.

- Advantages of Environment Variable overflow
  This means that if a variable is set to some exploitation code, that code is now available to jump to inside of a program. Now, the program must still be vulnerable to a stack overflow, but with an environment variable containing the exploitation code, the exploitation code can now be much bigger than before.

  There is an advantage here because one can create a very large NOP sled. This allows one a very high probability of landing inside the NOP sled with simply a high stack return address. This makes guessing the return address much easier and the attack theoretically easier to pull off.

  Therefore, the environment variable style of overflow seems to have the following advantages:
  1) The buffer one is overflowing can be small, smaller than the piece of exploit code. This is important if one cannot directly calculate or guess the size of the buffer, because in this case one is simple placing an arbitrary number of "new return addresses" at the beginning of the exploit string.
  2) The exploit code can contain an arbitrary number of NOPs plus the exploit code. No more complicated crafting of the exploit string.

3) The exploit string must only contain replicas of the desired return address. No more complicated crafting of the exploit string.

- How an environment buffer overflow attack works
  Given additional assumptions the environment buffer overflow attack works exactly like a classic overflow attack save it directs the return address to within an environment variable that contain exploit code.

  The environment buffer overflow attack has the following additional assumptions:
  1) Environment variables and their values are contained in the program's stack.
  2) These environment variables are located at the top of the stack.
  3) Code containing the environment variables may be executed.
  4) The value of an environment variable can be of an arbitrary size.

  With these assumptions (not all inclusive), the idea behind an environment buffer overflow attack is to place ones exploit code inside an environment variable. Then with the code inside an environment variable one modifies the return address of vulnerable code to return inside the value of the environment variable. This is possible because one assumes that all of the environment variables are contained in the top portion of the stack. Therefore one guesses a return address some where near the top of the stack. With this done, the buffer overflow with an address into an environment variable, which then runs some exploitation code.