Nullp0inter's Glorious Intro to Buffer_Overflows

morn.

The Basics: What Is A Buffer?

- Holds input to program
- Fixed size
- Think arrays
    - char buff[10] is buffer of size 10
    - usually lack bounds checking (*not always*)

```c
#include <stdio.h>
int main () {
   char buff[10];
   int overwriteMe = 0;
     scanf("%s",buff);
     if(overwriteMe != 0x1337){
        printf("Sorry, you failed\n");
        return 0;
     else {
        printf("Success!\n");
     }
     return 0;
 }
```

What goes on in the background

- Function calls create a frame on stack
    - Think of stack like “plates” of memory
        - holds buffer
        - holds vars
        - holds return addresses
- Buffer Overflow allows writing onto other parts of stack

So how does it work?

- Buffer has fixed size (usually)
- Other vars immediately follow buffer on stack
- Writing too much for unchecked buffer allows modifying the stack
- Easy to do with PYTHON :D

Important Stuff!

- Need to know how many bytes to deal with
- Addresses = 4 bytes, Ints = 4 bytes, Chars = 1 byte (language specific, look them up)
- 1 Byte = 2 Hex digits, e.g. 0xFF is 1 byte, 0xAABB is 2 bytes, etc.
- Different architectures use different "endianness" to read hex (next slide)
- Depending on compiler, stack smashing protection enabled by default
    - use compiler flag -fno-stack-protector

Very briefly on Endianness

- Order computers read hex
- two main kinds: consider 0xABCDEF12
  - Big Endian: human readable
    - 0xABCDEF12 -> 0xABCDEF12
  - Little Endian: Bytes are reversed
    - 0xABCDEF12 -> 0x12EFCDAB

The Code

- Take a look at example0 and example0.c
- Should be the following:

```
//Nullp0inter's Glorious intro to Buffer Overflows
//This one is just the basics, the buffer overfloweth with cyber riches yall

#include <stdio.h>

int main() {

    char buff[10];
    int overwriteMe = 0;

    scanf("%s",buff);

    if (overwriteMe == 0) {…
```

- You get the gist of it

So what does our stack look like?

- Call to function main builds a stack frame
    - pushes buff and other vars to stack
- int overwriteMe stored just below buffer
- *grows* downward to *lower* addresses
- Deals with hex representation
    - digits 0-9 and A-F (10-15)

The Stack:

Addresses	Contents
0xFFFFFA	buff[0]
0xFFFFF9	buff[1]
0xFFFFF8	buff[2]
. . .	. . .
0xFFFFF0	buff[9]
0xFFFFEF	overwriteMe = 0
0xFFFFEB	OtherStuff
0xFFFFE7	More Stuff
0xFFFFE3	Even More Stuff
0xFFFFDF	Probably a return address

So what happens when we input to the buffer?

- Buffer only holds so much data
    - If filled with more than it can hold, gets written past onto stack
- If we put 11 A’s in a 10 char buffer, we write 1 byte past buffer
    - notice value of overwriteME
- They don’t think it be like it do but it is

Addresses	Contents
0xFFFFFA	buff[0] = A (0x41)
0xFFFFF9	buff[1] = A (0x41)
0xFFFFF8	buff[2] = A (0x41)
. . .	. . .
0xFFFFF0	buff[9] = A (0x41)
0xFFFFEF	overwriteMe = A (0x41)
0xFFFFEB	OtherStuff
0xFFFFE7	More Stuff
0xFFFFE3	Even More Stuff
0xFFFFDF	Probably a return address

So what happens when we input to the buffer? -Continued-

- overwriteMe overwritten with 11th A

- Ascii / Unicode gets converted to hex to be stored

- Poor buffer means near full control of stack with overflows

Addresses	Contents
0xFFFFFA	buff[0] = A (0x41)
0xFFFFF9	buff[1] = A (0x41)
0xFFFFF8	buff[2] = A (0x41)
. . .	. . .
0xFFFFF0	buff[9] = A (0x41)
0xFFFFEF	overwriteMe = A (0x41)
0xFFFFEB	OtherStuff
0xFFFFE7	More Stuff
0xFFFFE3	Even More Stuff
0xFFFFDF	Probably a return address

- Malicious user can exploit control of stack to do as they please
  - bypass checks
  - escape program
  - get shell
- All of this is great but how exactly can it be done? There are a couple of methods but my favorite is....

PYTHON

yayyyyy lmao !!!

Using Python for Input

- can be done in single command and piped to program

    - Piping of the form: `<command> | <second command> | <third command>` … etc

- Can reverse “ and ‘ as long as used consistently

The specifics

- In general it starts like this:

```
    python -c “print ‘A’ * <buffer size>
+ <any extra stuff needed>” |
./<program binary>
```

- “Why don’t I just type everything manually or generate it and copy and paste?” Two main reasons:
    - buffer not always nice and small
    - not all hex has a character to it

example0

- requires overwriting integer overwriteMe
- goal is to make overwriteMe *anything* other than 0
- not so bad

example1

- In example1.c, notice overwriteMe checked against 0x1337

- Need to write past 10 char buffer with a *specific value* (0x1337)

- Take a minute to try it on your own

Solving example1

- Open a terminal (if you have not done so already) and type the command:

```
python -c "print 'A' * 10 + '\x37\x13'"
| ./example1
```

- So what is going on there?
  - buffer filled with 10 A's (buff = AAAAAAAAAA)
  - Hex value concatenated to A's and passed in Little Endian

Hints for example2

- example2 requires overwriting a function pointer
- figure out on your own
- use hints below to figure it out
  - `gdb`
  - `b main`
  - `p &<function name>`

Stack frame precedence

- Example in testing0 and testing0.c
- Typically buffer always first in stack
  - followed by vars in *reverse* order they were declared
  - set to print values, mess around a bit

Extremely Useful Tools

- Several extremely useful tools for overflow, ROP, and reversing:
  - GDB - GNU debugger to step through execution
  - peda - python based gdb extension to show assembly, registers, stack, etc during execution (nice colors)
  - radare2 - disassembler and more, get it from GitHub and *not* from apt repos (repo build severely out of date)
    - - can generate patterns as well (ragg2 -P <len> -r)

Resources

- gera’s insecure programming
- Hacking the Art of Exploitation
- Smashing the Stack for fun and profit by aleph1 (Phrack Mag)