Name: Sean Bartholomew
Date: 28 May 28, 2016
Current Module: Assembly
Project Name: Bomb

Step 1: Create a copy of the file.

Step 2: Hexdump –C ./bomb

The first line in the hex dump is: 74 45 4c 46 … . E L F
This shows that the file is executable.

Step 3: Strings ./bomb

I ran strings in an attempt to learn more about the file without having to execute it. Below are the results, along with a brief description of why they may be important.

/lib64/ld-linux-x86-64.so.2
This tells us that the program is likely compiled for a 64-bit Linux system.

bomb.c
stages.c
start.c
stage_secret.c
stage_8.c

These strings along with the below make me believe that the program was written in the C language, specifically conforming to C99

puts	puts@@GLIBC_2.2.5
memalign	memalign@@GLIBC_2.2.5
__stack_chk_fail	__stack_chk_fail@@GLIBC_2.4
__isoc99_sscanf	__isoc99_sscanf@@GLIBC_2.7
Stdin	stdin@@GLIBC_2.2.5
_exit	_exit@@GLIBC_2.2.5
	exit@@GLIBC_2.2.5
fgets	fgets@@GLIBC_2.2.5
strlen	strlen@@GLIBC_2.2.5
memcpy	memcpy@@GLIBC_2.14
mprotect	mprotect@@GLIBC_2.2.5
malloc	malloc@@GLIBC_2.2.5
	free@@GLIBC_2.2.5
getenv	getenv@@GLIBC_2.2.5

stderr	stderr@@GLIBC_2.2.5
ptrace	ptrace@@GLIBC_2.2.5
strncat	strncat@@GLIBC_2.2.5
strchr	strchr@@GLIBC_2.2.5
fprintf	fprintf@@GLIBC_2.2.5
	printf@@GLIBC_2.2.5
syscall	syscall@@GLIBC_2.2.5
strcmp	strcmp@@GLIBC_2.2.5

The strings above are all referencing standard C function calls and will likely be run in the course of the program.

Nice try, but such a stage does not exist.UH
BOOM
You have died
You were able to defuse %d stage%c before exploding.
Have a nice day.
Bomb disarmed!
You live!
Stage 1:
Stage 2:
Stage 3:
Stage 4:
Stage 5:
Stage 6:
Stage 7:
Stage 8:
Stage ?:

These strings are likely communication with the user. Considering the above references to puts(), printf(), and fprintf() I expect them to be printed to the user during the course of the program.

%u %u %u %u %u
%c %d %c
%d %c %d
?%d %c %d

Format strings. These will be used to either print some kind of data to the user or to evaluate user-input. The previous mention of fgets() indicates the latter.

stage_7
stage_8
stage_3

stage_1
stage_5
stage_4
stage_2
stage_6
stage_defused
stage_secret
stage_super_duper_secret
main
stage_8_input.txt
swordfish
USER
hidden_message
name_length
username
filename
hidden_phrase

These strings are interesting because they do not appear to be related to standard function calls. These strings are likely variables and / or function names within the program.

Step 4: objdump –M intel –d ./bomb | less

Utilize objdump to examine the program in assembly language. At this level I can start to understand what the program is doing and how it works. The program appears to be broken up into 14 functions: main() stage_diffused(),stage_1(), stage_2(), stage_3(), stage_4(), stage_5(), stage_6(), stage_7(), stage_8(), stage_secret(), stage_super_duper_secret(), ___(), _start()

Step 5: gdb ./bomb
tui enable
layout asm
layout regs

After running strings the next step is to parse through the function in gdb. This is done to prevent the program as much as possible from doing anything malicious. However, gdb says that the registers are unavailable. The program clearly is doing something to prevent gdb from functioning properly.

Step 6: Google

I attempted to see if there was a reason that gdb wasn't functioning correctly. My search turned up: http://reverseengineering.stackexchange.com/questions/1935/how-to-handle-stripped-binaries-with-gdb-no-source-no-symbols-and-gdb-only-sho . The site shows how to get gdb to set a breakpoint at the start of execution of the file.

Step 7: gdb ./bomb
info file
break *<entry point of program>
start

Gdb now functions properly. Stepping through the program, it calls ptrace() in a loop, and seg-faults. The program is not intended to be run in a debugger.

Step 8: gbd ./bomb
break *<entry point of program>
start
jump main -- suggestion from Primm.

Gdb is now fully functional and is no longer looping to a segmentation fault. I can now step through the program to determine what the different functions are doing.

Functions

_start():
Normally a function which is auto-generated by the compiler. This function has been specifically written to segmentation fault if the program detects that it is running in a debugging environment. It utilizes ptrace() to determine if it is in a debugger.

main():
main calls stage_diffused().

stage_diffused():

This is the controlling function of the program. It's primary purpose is to prompt the user for input, and call other functions to manipulate that input. If the called functions return value is 1, the program continues to the next "stage." Any other return value results in the program terminating with a message

BOOM
You have died
You were able to defuse %d stage%c before exploding.

int stage_1(char * input): -- solved by Iracane

Stage 1 compares the user input to the string at 0x401900 utilizing the c library function strcmp(). If the strings are the same the return value is 1. The string is "swordfish".

int stage_2(char* input): -- solved by Paradis

Stage 2 utilizes the c library function getenv() to determine the username of whomever is running the program. The input string is compared to the username with strcmp(). If the strings are the same the return value is 1.

int stage_3(uint input): -- solved by Torres

Stage 3 concatenates the username and the input string. The function finds the length of the resulting string (length) utilizing strlen(). The input value is then divided by length and divided gain by 4 (value). Value is then multiplied by the constant 0xAAAAAAAB. If the result overflows into the Dx register, the function will exit successfully.

Min values	strlen(<username>)
270	0
390	1
504	2
672	3
864	4
1080	5
1320	6
1584	7
1872	8
.	.
.	.

int stage_4(char* input) -- solved by Simpson

The input string in Stage 4 is checked utilizing scanf(). It must be 5 unsigned int characters. The function works in a loop:

```
int sum = (int)<first character of username>
int n = 0;
while ( n <= 4){
        if (input[n] > sum){
                sum += input[n];
        }
        else{
                goto fail;
        }
```

```
            n++;
        }
```

If the loop completes for all n, the function returns a success code.

int stage_5(char* input): -- solved by Simpson

Stage 5 expects the input string to be in the form of %c %d %c. The function utilizes scanf() to ensure the string is formatted correctly. The following algorithm is applied to the input:

```
int a = (int) <first char argument>
int b = (int) <second char argument>
int d = (int) <integer argument>
int sum = 0;
int count = 0

while ( count > strlen(<username>) ){
    switch ( username[count] ){
            case a, e, i, o, u, y:
                sum -= a + (int)username[count]
                count++
                break
            case b, c, d:
                sum -= (int)username[count]
            case f, g, h:
                sum -= (int)username[count]
                count++
                break
            case j, k, l, m, n:
                sum += b - (int)username[count]
                count++
                break
            case p, q, r, s, t:
                sum -= b + (int)username[count]
                count++
                break
            case v, w, x, z:
                sum -= b
                count++
                break
    }
}
if sum == d{
    return <success>
```

}
return <fail>

int stage_6(char* input) : -- solved by Bart

Stage 6 expects the string to be in the form of %d %c %d. The input is verified by scanf(). The function passes the two integer values to the ___() and does the following comparison:

int x = <input1> + <input2>
int y = (int) (___(input1, input3)& input2)

If x == y the function returns success, else failure. For success, both digits must be the same. The successful return from ___() is the first character of your username.

int ___(int a, int b): -- solved by Bart

This function operates recursively, first it checks to see if the integers are equal to zero, if not it decrements a, sets a equal to b, and b equal to a. The loop will continue until either both arguments are zero, or a segmentation fault occurs. (I am not sure why it segmentation faults I did not look into that). If both arguments become zero the first character of your username is returned.

int stage_7(char* input): -- solved by Iracane

I do not know the expected behavior of stage 7. If input is NULL, the stage is successfully bypassed. I do not believe this is how you get into the secret stages as some of my classmates have postulated. There is only one exit point for the function, and it is not into the secret stages.

stage_8() -- unsolved

I do not know what this function does. I do know that a successful exit from the stage will return a 1. Everything else is a failure.

stage_secret() --unsolved

This function is called by the stage_diffused() function after stage_8(). I do not know what this function does. The objdump shows that the expected success code is 1, however there is no way for the function to return a 1. There is only one return statement in the function and the lines immediately preceding it set the return value to 0.

```
4016c8:     b8 00 00 00 00      mov    eax,0x0
4016cd:     83 e0 01            and    eax,0x1
```

```
4016d0:    c9              leave
4016d1:    c3              ret
```

However, there is another possible way out of the function there is a jump to the value of RAX. This jump could potentially be to the super_duper_secret() function as there is no explicit call to in anywhere in the objdump.

```
4015c9:  89 d7             mov    edi,edx
4015cb:  ff d0             call   rax
```

At this point the value of RAX would need to be 0x4014ef for my theory to be correct.

super_duper_secret() --unsolved

I do not know what this function does. I do know that there are no explicit calls to it. There is the possibility of getting to the function through secret_stage() however. This function has no return call. It does have several jumps to the secret_stage. The two must function together somehow.