CS161 Final Project Part 1

Fall 2009

Introduction

In this first part of the project, you will find buffer overflow vulnerabilities, craft exploits to inject and execute your own code, bypass non-executable-stack protection mechanism, think about security issues beyond buffer overflows, and fix the security problems you identify.

Because buffer overflow exploits depend on exact memory layout, a given exploit is unlikely to work if the vulnerable binary is complied by another compiler or started on another system or in another environment. To avoid this problem, the vulnerable web server will be running in a virtual machine. Your exploits will be embedded in HTTP requests that can be sent to the web server from anywhere.

Our many thanks go to Nikolai Zeldovich who is the author of most of the code and the setup used in this project.

Getting Started

To get started, you need to download and install VMware Player that is freely available for Windows and Linux. Once you have the VMware Player set up, download a virtual machine image. The two virtual machine images `cs161-vm.7z` and `cs161-vm.tar.gz` are available at http://www.eecs.berkeley.edu/~igor/cs161/. Both images are identical. 7z is smaller, but requires more time to unpack.

The virtual machine (vm) is a minimal Ubuntu installation (without a graphical interface). Besides the `root` user, the vm has an `httpd` user. The passwords for both users are `cs161proj`. The httpd user has also been given root privileges so that it can run any command that requires root privileges by prefixing the command with `sudo`. In particular, if you want to install a package while logged in as `httpd`, you can execute `sudo apt-get install <package-name>`.

To work in the vm, you can either login from the console or better ssh into the machine and work from your terminal. To ssh into the vm, you will need its IP address. You can learn the IP address by executing `/sbin/ifconfig eth0` in the vm.

The home directory of httpd user has a `proj1` directory and the tar file `proj1.tar.gz` containing the same files as in the directory. If at any point in time, you would like to have the original files, you can untar them from this archive or get this archive from http://www.eecs.berkeley.edu/~igor/cs161/.

The first thing to try to is to make sure all is working fine. First, try to compile the web server:

```
    httpd@cs161:~$ cd /home/httpd/proj1
    httpd@cs161:~/proj1$ make
    gcc httpd.c -c -o httpd.o -m32 -g -std=c99 -fno-stack-protector -Wall -Werror
-D_GNU_SOURCE=1
    gcc httpd.o -o httpd -m32
    cp httpd httpd-exstack
    execstack -s httpd-exstack
    cp httpd httpd-nxstack
    execstack -c httpd-nxstack
```

```
    gcc shellcode.S -c -o shellcode.o -m32 -g -std=c99 -fno-stack-protector -Wall -Werror
-D_GNU_SOURCE=1
    objcopy -S -O binary -j .text shellcode.o shellcode.bin
    httpd@cs161:~/proj1$
```

Then, run the web server:

```
    httpd@cs161:~/proj1$ ./httpd 8080 .
```

This command will start the web server listening on 8080 and serving requests with current directory as the web home directory. Now, go to any web browser that can reach the vm and try to connect the the server by specifying the address like `http://192.168.19.129:8080/` where 192.168.19.129 should be substituted with the IP address of your vm.

Contents of the proj1 Directory

- **httpd.c** is the source code of the web server. You can play with the code as you wish, but your exploits has to work on the original version of the web server.
- **clean-env.sh** is a simply shell script that you can use to always start the web server in the same, clean environment. This is useful to ensure that httpd behaves identically from start to start. You can use it like this - "`./clean-env.sh httpd-exstack 8080 .`"
- **Makefile** is contains rules for building the two versions of httpd (httpd-exstack with executable stack and httpd-nxstack with non-executable stack), as well as rules for compiling and extracting the shellcode assembly program. As shown above, to compile you just need to execute `make` and to delete all the outputs you need to execute `make clean`.
- **exploit-template.py** contains a simple python script that sends a bare-bone HTTP GET request to the server:port you specify as command line options. This script can be useful for you to send your hand-crafted HTTP requests by modifying the script. If you choose not to use this template, you will need to give us another script/program that will send your crafted HTTP request.
- **index.html** is our basic home page
- **app.py** and **applogic.py** are python scripts that generate our dynamic web content
- **shellcode.S** is the shell code from the Aleph One's tutorial that brings up a shell. You can use this shell code to write your initial exploit without having to worry about the shell code. Then, you can write your own shell code that deletes a file (more on this later).
- **.gdbinit** is an initialization file for gdb. It is automatically read and executed by gdb if it starts in the same directory. It currently contains a single command, "unset env", which serves the same purpose as clean-env.sh. If you find yourself restarting gdb many times and having you input the same command over and over again, you can put your commands (one command per line) in this file.
- **answers.txt** is an empty file for your writeup.

Several important files appear after compilation:

- **httpd-exstack** is a version of httpd with stack marked executable
- **httpd-nxstack** is a version of httpd with stack marked non-executable
- **shellcode.bin** is the compiled binary code of shellcode.S so that you can just open it with your favorite binary file viewer (you can open it in emacs and then enable binary mode with "M-x hexl-mode" command) and copy the bytes into your exploit script.

Questions and Requirements

Question 1: The web server source code (httpd.c) has a number of buffer overflow vulnerabilities. Find

as many vulnerabilities as you can. For each vulnerability you identify, point it out (e.g. give the line number) and explain in words how the buffer can be overrun. Be specific.

Question 2: For each vulnerability you found in question 1, write a separate script (e.g. by copying and modifying exploit-template.py) that sends one or more HTTP requests and exploits this vulnerability. You don't have to inject or execute any code. Most probably, your exploit script will just crash the server. In your write up, tell us what script exploits what vulnerability and how we should run your scripts. If for any vulnerability, you believe that it cannot be exploited in this fashion, describe why? Your exploit will probably work on any version of httpd, but to be specific, we will test them on httpd-nxstack.

Question 3: Pick one vulnerability you found in question 1 and exploit it in httpd-exstack to delete the `/home/httpd/grades.txt` file. Similar to question 2, you should write a separate script that exploits this vulnerability. In your write up, explain how your script exploits the vulnerability and tell us how we should run your script. You will need to write your shell code (or derive it from shellcode.S), compile it and extract the binary code to include it into your script. To compile and extract the binary code, you can either name your shell code shellcode.S and run make, or just look into the Makefile for the commands and run them yourselves. To delete the file, you can either use /usr/bin/unlink or /bin/rm.

Here are some hints on how to get started. First, read the Aleph One's tutorial on stack smashing at http://www.phrack.com/issues.html?issue=49&id=14#article. After reading and thinking about it, you should have a pretty good idea on how exactly buffer overflows work and how to write a shell code. To actually get started with crafting an exploit, you can follow the following steps. First, run httpd-exstack in gdb ("gdb httpd-exstack", then in gdb prompt "set args 8080 .", then set a break point and run). Look at the stack where you want to overwrite the buffer and decide how you can capture the execution of the program. Once you can get hold of the program flow, work on your shell code (or try Aleph's first) and on injecting it and jumping to it.

You should definitely get familiar with gdb. Especially useful commands are probably p, x, s, si, n, disas, list, break, bt, frame, help. To get a quick help of a commad cmd you can type "help cmd". You can also change any (writable) memory location with something like "`set {int}0xBFFFF09a = 161`".

Question 4: In this question, you will use another method of exploiting buffer overflow attacks. In the previous question we relied on the assumption that the stack is executable. However, most modern operating systems mark the stack non-executable by default. The httpd-nxstack is the version of httpd with stack marked non-executable and it is the version that you will be using in this question.

Since we cannot place code on the stack and execute it, we might look for some code that is already in the program address space that we can use to achieve our goals. The standard code repository to look for such ready code is the C library that is well documented and almost always available. An attack that uses a standard C library function simply prepares the arguments for the function on the stack and makes the program jump to this function. This type of attacks is usually called *return-to-libc* and is described in greater details in this article http://www.milw0rm.com/papers/31. You probably won't be able to follow the steps in the article precisely because when I tried it my gcc had a different stack structure for the main function's frame, but the general description is certainly applicable.

Choose a vulnerability that is different from the vulnerability you chose in question 3 and exploit this

vulnerability in httpd-nxstack to delete the `/home/httpd/grades.txt` file. You should have a separate script to do launch this attack as well as a write up explaining how your attack works.

Question 5: In this question you are asked to find any security problems (not necessarily buffer overflows) with the web server. For each problem you find, write an explanation of the problem, what the attacker can achieve, and how you can try to fix it.
One way to approach finding general problems is to look in all the places where inputs from the outside world are used in the code. Try to imagine weird inputs that the code author would never expect, but the attacker can happily send.

Question 6: In this question, modify httpd.c to produce httpd_safe.c that fixes all the problems you found in question 1 and 5. Comment you fixes in the code. To distinguish your comments from our original comments you can either delete all of our comments or prefix your comments with something like "/* FIX: */" so that it is easy for us to find your fixes.

General Requirements:

- All of your scripts should be runable in the vm. We will do all our testing inside the vm. In case you need to install any packages to run your scripts, please write a script that does all the installation and explain how to run it in you write up.
- For testing, we will run the web server in the vm as follows. Make sure you exploits work when the web server is started in this way.

```
httpd@cs161:~$ cd /home/httpd/proj1
httpd@cs161:~/proj1$ ./clean-env.h httpd-xxstack 8080 .
```

  where `httpd-xxstack` is either `httpd-exstack` or `httpd-nxstack` depending on the question.

Deliverables and Points:

Question	Deliverables	Points
1	Write-up (in answers.txt)	3 pts per vulnerability
2	A script for each vulnerability and a write-up indicating what script exploits what vulnerability and how to run it	3 pts per script
3	Script, shell code, write-up	15 pts
4	Script, write-up (shell code if needed)	15 pts
5	Write-up	5 pts
6	httpd_safe.c with comments for each fix	10 pts

Submission Instructions

You need to email your submissions (one submission per group) to Igor by 9am on Monday, November 23rd. Your email must have "cs161 project part 1 submission " as the subject line (no capital letters please). Your email must contain only one tar (or zip) file named proj1_gN.tar.gz (or proj1_gN.zip), where N is the number of your group. The archive should contain (1) answers.txt containing all the write-ups, (2) a number of executable scripts (see above), all documented in answers.txt, (3) shell code for question 3 (and for question 4 if you used any), (4) fixed web server source code with comments in httpd_safe.c.

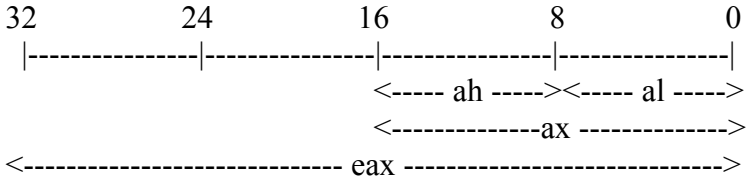
x86 Assembly

As part of the project you will write the code to be injected and executed. The most reasonable way to

write it is to write it in assembly. Therefore, you will need to learn some x86 assembly. There are plenty of resources on x86 assembly on-line and you should look for them. In this section, we describe some important points that you should know.

Basics. There are 6 general purpose registers in x86 called eax, ebx, ecx, edx, esi, edi. The last two of these have a different name because they are usually involved in vector instructions, but for our purposes, they are all the same. Then, there are two registers ebp and esp that help keep track of the stack. ebp points to the base of the frame (i.e. it is a frame pointer) and esp points to the last used byte of the stack (i.e. it is the stack pointer). Then, there is eip, which is the instruction pointer (i.e. program counter). Finally, there is the eflags register that holds different flags such as whether the last arithmetic operation returned zero or not (you should not need this register). There are more registers such as control registers whose names start with "cr" and segment registers whose names end in "s". You don't need to know about them.

All registers are 32 bits and different parts of a register have their own names. For example, eax is the whole 32 bites, ax is lower 16bits, al is the lower 8 bits, ah is the higher byte of ax



Different Syntaxes. There are two widely used assembly language syntaxes: the Intel and the AT&T syntax. GNU project (and its programs gcc, gdb, as, etc) use the AT&T syntax. Thus, if you choose the standard Linux tools to work with, you will be using the AT&T syntax. However, if you choose other tools or come across resources that use the Intel syntax, you should be aware of its existence.

The article at http://asm.sourceforge.net/articles/linasm.html briefly describes the two syntaxes and their differences. Here are some important points of AT&T syntax:

- The source always comes before the destination.
- Instructions have prefixes that specify the size of the operation. For example, `movb` will move one byte, `movl` will move 4 bytes.
- Registers are prefixed with % and immediate data (constants) are prefixed with $.
- AT&T syntax has obscure syntax for computing effective addresses. For example, if you want to compute an address of an element in a structure in an array of such structures, you will do it using the base pointer (the beginning of the array), the index of the structure, the size of the structure, and the offset of the element in the structure: `base+index*size+offset`. In AT&T syntax, this computation is written as `offset(base,index,size)`. For example, the instruction "`leal (%ebx,%ecx),%eax`". Will do `eax=ebx+ecx` (because size defaults to 1 when not given).

Non-obvious useful instructions.

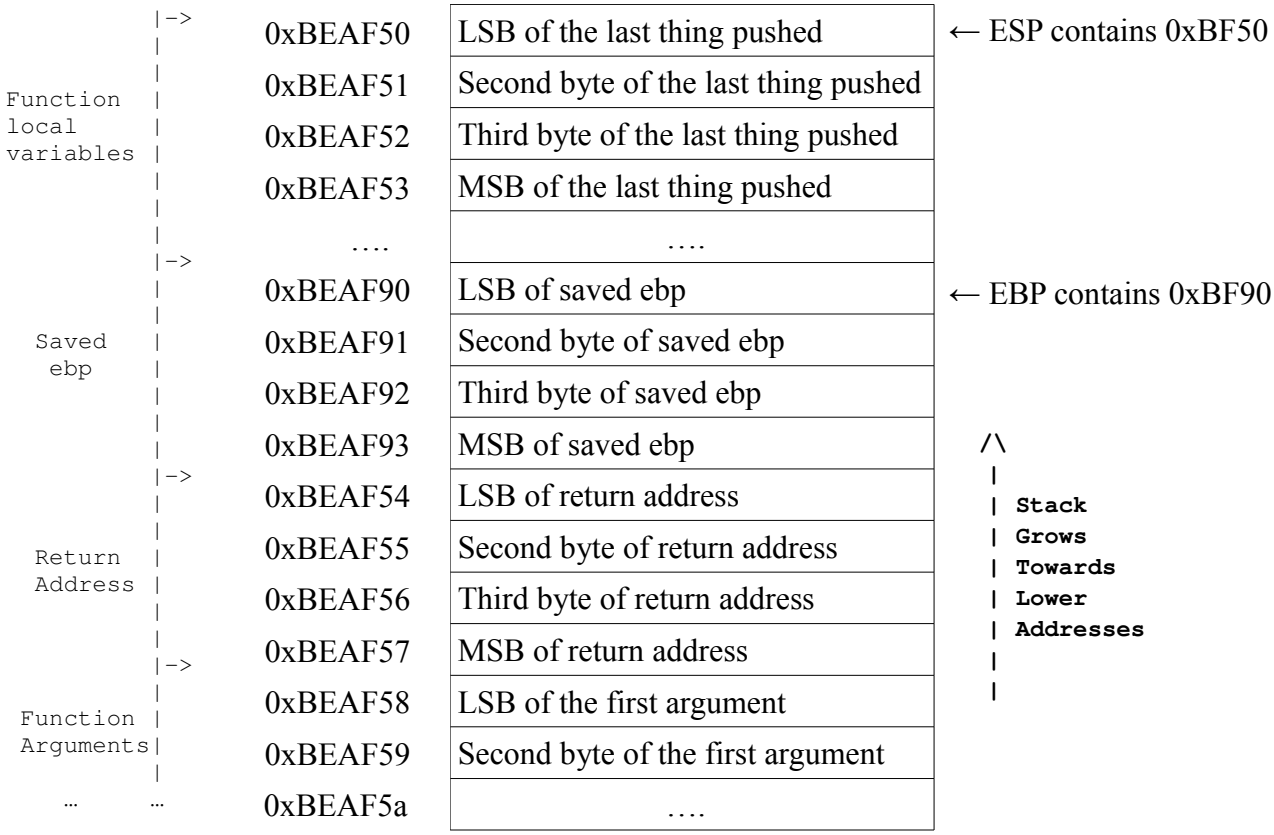
- call addr – pushes the address of the next instruction on the stack and sets the eip register to addr (thus, effectively jumping to addr)
- ret – does the opposite of call. It pop 32 bits from the top of the stack, treats them as an address

and sets eip to this address

- leal – Load Effective Address. Letter "l" at the end is the suffix for "long" (see above). This instruction is usually used to compute the address on which we will be doing operations next.

Stack

Recall that stack starts at high addresses and grows towards low addresses. The esp register holds the address of the last used byte (i.e. the lowest memory address occupied by the stack). The ebp register holds the address of the least significant byte of the saved frame pointer. This is the standard calling convention. The compiler might not follow this convention in some special cases. Also, recall that x86 is little endian, which means that the least significant byte is in the lowest address. With all these piece of information, a usual stack frame at the byte level looks like this (LSB/MSB stands for least/most significant byte):



Misc FAQ

Can I run httpd on my own machine? The short answer is you can, but you are on your own. Note that most likely the memory layout of httpd will be different if you run it on your own machine and the exploits you develop there will not work if httpd is run in the VM (that is how we will test it). If you decide to run it on your own machine, you will need to disable address space randomization (`sudo sysctl -w kernel.randomize_va_space=0`) that by default changes the address space layout each time a program is started.

Why do you ask us to write a shell code? Because it will give you a practical understanding of assembly that is helpful in understanding software security.

How long will this project take? It can be a rather time consuming project, especially if you are not familiar with assembly and gcc. Given this, please start early and get to your GSI if you get completely stuck and cannot make progress. However, understand that unless your question is rather simple, GSIs cannot answer it via an email.