CSE 484/584M

Lab 1a/b: Buffer Overflows

Lab 1a Due (Sploits 1-3): Friday, January 20, 5:30pm
Lab 1b Due (Sploits 4-7): Wednesday, February 1, 5:30pm
Signup Form:
[https://docs.google.com/forms/d/1aqlm9lxiVdVjKCON5JO_anXGLG_D3UCWKaCUAVwOf5M]

[Make sure you are signed into your UW Google account]

Goal

- The goal of this assignment is to gain hands-on experience with the effects of buffer overflow bugs. All of the work must be done on the machine `cse484.cs.washington.edu` (see instructions below for connecting).
- You are given the source code for seven exploitable programs, whose binaries are stored in the "labbin" directory (/labbin/target1, ... , /labbin/target7). Each target program [i] is installed as setuid hax0red[i]. Your goal is to write seven exploit programs (sploit1, ..., sploit7). Program sploit[i] will execute program /labbin/target[i], giving it certain input that should result in a shell run with the same permissions as user hax0red[i]. If target[i] had setuid root then sploit[i] would result in a root shell. We don't do that in this case for obvious security reasons, so instead you get the permissions of the hax0red[i] user.
- The skeletons for sploits 1 through 7 are provided in the ~/sploits/ directory. Note that the exploit programs are very short, so there is no need to write a lot of code here.
- Sploits 1-7 are required. Sploit 8 is extra credit.

The Environment

- You will test your exploit programs on a remote machine running Ubuntu Linux hosted at the domain `cse484.cs.washington.edu`
- To connect to the machine, each group must first join a Canvas Group under the **"Lab 1 Group" group set**. In Canvas, go to the "People" tab on the left and select the "Lab 1 Group" tab near the top (or you can also just go to this link (https://canvas.uw.edu/courses/1610855/groups#tab-162414), and make sure that you and your teammates join the same group. Then each of you should respond to the Lab 1 Sign-Up Google Form [https://docs.google.com/forms/d/1aqlm9lxiVdVjKCON5JO_anXGLG_D3UCWKaCUAVwOf5M], make sure to log in with your **@uw.edu** Google Account) with your ssh **public** key (from a key pair that you control; strongly encourage using ed25519 keys).

- Expect some delay (~10 minutes) between sending the information and account creation, so please answer the form early and plan accordingly. Please double check that your answers are free of typos to minimize any delays. Also, do **NOT** send the private key (that should remain only on trusted machines). **Each group member** needs to submit a single public key to access cse484 -- see the "Misc" section later in the lab description about how to grant access later to additional devices without having to copy private keys.
- After at most 10 minutes, you should be able to SSH into cse484. The username for your team will be **cse484-23wi-X**, where X is your group number on Canvas (in the "Lab 1 Group" group set, not zero-padded). See the next section for instructions on SSH-ing with your RSA key. Contact a TA if you are not able to log in after 10 minutes.

Here are is an example of a public key (generated with `ssh-keygen -t ed25519`):

```
ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIOHK0eNCjWCsVX/otyjCFadkLe89W9Ep4Mqk00R
0ODdp user@localhost
```

Generating Key Pairs

Linux/Mac:

To generate a key-pair run the following (it is strongly suggested to use a passphrase):

ssh-keygen -t ed25519 -f <key_name>

To use ssh (after we have created your account):

ssh -i <path_to_private_key> <username>@cse484.cs.washington.edu

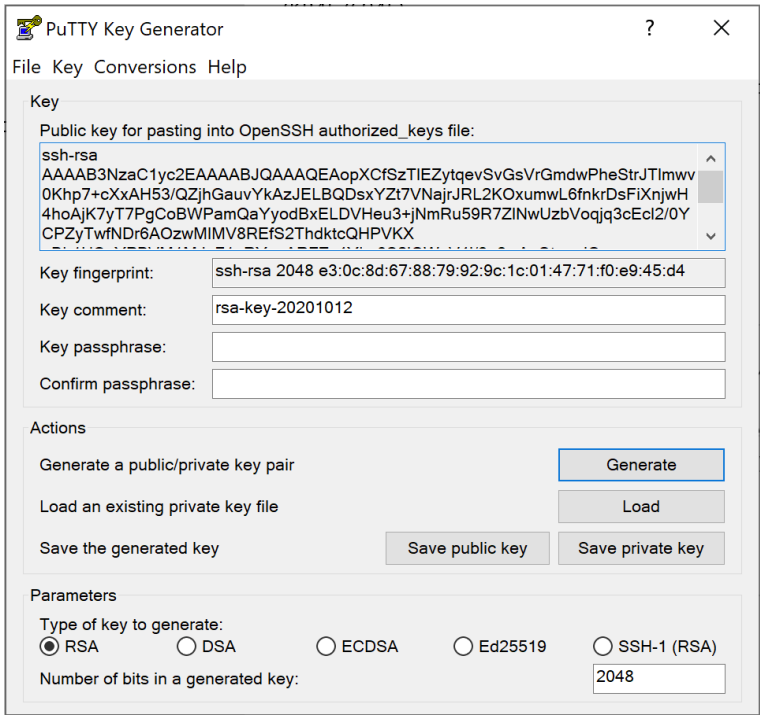
Windows:

Option 1:
You can generate the key-pair the same way as Linux/Mac using PowerShell or Windows Subsystem for Linux (WSL).

Option 2:
To generate the key-pair use PuTTYgen. It comes installed with PuTTY.

1) Open PuTTYgen
2) Select the type of key as EdDSA (or RSA)
3) Click Generate and move the mouse around to generate entropy
4) (Optional but recommended) Enter a Passphrase + Confirmation of Passphrase

5) Click save the private key
6) Copy the text of the public key to post/email from the box at the top



Format conversion:

If the public key you get looks like the SSH2 format shown below, you will need to convert both public and private keys to the OpenSSH format.


```
---- BEGIN SSH2 PUBLIC KEY ----
Comment: " user@localhost"
AAAAB3NzaC1yc2EAAAADAQABAAABAQDTKPi45wxeSezgO5JmG8HiuAQH6R3kqQTe
OeTbntWxliiClrahwlnkv26PAIaQKNdRbVH1fgX9kyUfsdj5JAvvNFuxpfY+GVVZKFI5M3Cuz
AynIymBjqnDn6Auq+tuSl8O4osb/0L9zDeQzOxQ+ed6iVDuPPkBLoX+XyuNUyYKV46xCIHO
S6ao+6CkZXhp4VTz4LUvb1s8DIUcaD8/bbigxxZH3eKRQH2arV9AqP1LoC2T3azLTkHvCrcI
mpjVW/pxf5+nbkRb1SSkkHFvFPdd+0us12yGOp1xBbo2kuKWSdcBgd4eiGHQsO+VWi23R92
bcOh/DxRZumdMyaDBMGY/
---- END SSH2 PUBLIC KEY ----
```


To convert the private key: go to the "Conversions" menu in PuTTYgen, and click on "Import key." Select your .ppk private key file. Then go to "Conversions" again, and click on "Export OpenSSH key." Save the exported private key file and you should be able to log in with it.

To convert the public key: use command `ssh-keygen -i -f ssh2.pub > openssh.pub`

To ssh (with PuTTY):

On the left side, select Connection->SSH->Auth. In this pane, browse to your private key, and then login as usual.

Enter <username>@cse484.cs.washington.edu in the Host Name field (where <username> should resemble cse484-23wi-X), and 22 for the Port field. For Connection Type, make sure you select SSH.

You may want to save the session for a quicker login next time. (Note, if you generated your ssh key pairs using Linux and you want to use it in windows, you will need to use PuTTYgen to convert it from .pem to .ppk before using it)

The Targets

- The targets are stored in /labbin and their corresponding sources in ~/sources/. You are free to study the source code of each target. **DO NOT** recompile the targets!
- Your exploits should assume that the compiled target programs are installed in /labbin. Do not move the targets.
- Each target[i] is setuid hax0red[i], which means that they run as hax0red[i] regardless of who runs it. The one exception is when they're run under a debugger. Allowing users to debug a setuid executable is a security flaw, so setuid programs temporarily lose their setuid-ness under a debugger. This means that you can only get a hax0red[i] shell when your sploits are ran outside of gdb. However, if you get a user shell inside gdb, you should get a hax0red[i] shell outside of gdb.

The Exploits

The ~/sploits/ directory contains the source for the exploits which you are to write, along with a Makefile for building them. Also included is shellcode.h, which gives Aleph One's shellcode.

The Assignment

You are to write an exploit for each target #1-7. Each exploit, when run on the remote machine, should yield a hax0red[i] shell (/bin/sh). To confirm this is working, run the command `whoami` in the shell, and you should see the hax0red[i] user.

Extra Credit

Target8 is extra credit! You can see that the source code is exactly the same as target0, except this time, the stack is not executable. You might want to try a return2libc attack. Here's a good tutorial for it: RET2LIBC (starting from page 52).

Hints

- Read Aleph One's "Smashing the Stack for Fun and Profit." Carefully! We also recommend reading Chien and Szor's "Blended Attacks" paper. These readings will help you have a good understanding of what happens to the stack, program counter, and relevant registers before and after a function call, but you may wish to experiment as well. It will be helpful to have a solid understanding of the basic buffer overflow exploits before reading the more advanced exploits.
- Read scut's "format strings" paper. You may also wish to read http://seclists.org/bugtraq/2000/Sep/214.
- gdb **(or cgdb, which gives you a view of the source code you're debugging; we also have gef (https://hugsy.github.io/gef/) gdb installed, if that's your thing, just add it to your .gdbinit with `echo "source /usr/share/gef.py" >> ~/.gdbinit`)** is your best friend in this assignment, particularly to understand what's going on. Specifically, note the "disassemble" and "stepi" commands. You may find the 'x' command useful to examine memory (and the different ways you can print the contents such as /a /i after x. If you use gef, telescope is a really useful command to check the information stored on stack. In gef, if you lost the original nice display, you can use context to tell it to reprint it.) The 'info register' command is helpful in printing out the contents of registers such as ebp and esp. The 'info frame' command also tells you useful information, such as where the return EIP is saved.
- A useful way to run gdb is to use the -e and -s command line flags; for example, the command `cgdb -e sploit3 -s /labbin/target3 -d ~/sources` in the vm tells gdb to execute sploit3, use the symbol file in target3, and the -d shows you the source code of the target as you step through it. These flags let you trace the execution of the target3 after the sploit has forked off the execve process. When running gdb using these command line flags, be sure to first issue 'catch exec' then 'run' the program before you set any breakpoints; the command 'run' naturally breaks the execution at the first execve call before the target is actually exec-ed, so you can set your breakpoints when gdb catches the execve. Note that if you try to set breakpoints before entering the command 'run', you'll get a segmentation fault.
- If you wish, you can instrument your code with arbitrary assembly using the asm () pseudo function.
- Make sure that your exploits work within the remote environment we provided.
- **Start early!!!** Theoretical knowledge of exploits does not readily translate into the ability to write working exploits. Target1 is relatively simple and the other problems are quite a bit more complicated.
- Find more FAQs answered here: https://courses.cs.washington.edu/courses/cse484/23wi/assignments/lab1-faq.pdf

Warnings

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. Addresses on the exploit's stack can change based on how the exploit is executed (working

directory, arguments, environment, etc.); in our testing, we do not guarantee to execute your exploits as bash does. **You must therefore hard-code target stack locations in your exploits.** You should **not** use a function such as get_sp() in the exploits you hand in.

Deliverables

- You may work in groups of up to **three** people. Make sure your submission includes the name of all your group members, and that you self-organize into your Lab 1 Group via Canvas.
- **In a bid to get you to start early, sploits 1-3 are due early as Lab 1a (see header).**
- We provide a `handin.sh` script that will make a copy of your current sploit code, and print out a list of hashes for those sploits. You can find that script here, and you should copy it to and run it from your home directory on cse484. Here is how you can get it to work:

```
$ wget https://courses.cs.washington.edu/courses/cse484/22au/assignments/handin.sh
$ chmod u+x handin.sh
$ ./handin.sh
```

- You will hand in those hashes, and since we have access to your remote home directory, you won't need to submit any code. However, to let us know when you're done, please submit a text file named

  `<netid member 1>_<netid member 2>_<netid member3>.txt`

  With the output of the handin.sh script, which is automatically put in a log file as indicated by the script. It should look like this:

```
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit0.c:
11f5fbce21e8b67baf9abfdabf0a726e16cbfdef424d640946b7dcc1fff45a82
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit1.c:
132677d732c935f22092f94f11e99406fe477a2b6e466c9850a3d7d66fe567e6
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit2.c:
42a2a96d32606417e989fe6cd0aa7156b87ebba58e780890bcde42b73b8f464e
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit3.c:
cfba30c73b1401c7c7f13d064fba082b39a67a9a5b9ddf0b8fdb1d251571b1f5
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit4.c:
da7b2d5f3495f7ea3a13d8f1e22d3e190c5e984c5451f1f74e1bd8e1709ee01d
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit5.c:
98b9abfe43a056aa0a430087d0ad913ab62e50605055847a3ec2a4b30d95171a
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit6.c:
ef3a3f3b54eb6ae62438ef93aacbb4d6a735afea33a7a8b7763603f6e23110de
/home/$USERNAME/turnins/sploits_22_20_04_13:20:16/sploit7.c:
ada265ca342ac20dd9a75b62d298852e89bf01048468dbcc55c7ca3c845dafd4
```

- Turn in your text file online via the Canvas assignment.

Misc

- **Please try to access the remote machine early and let us know if you have any problems! Feel free to check in with us if you don't have access within 24 hours of sending your public key. But please do allow 24 hours :)**

- If you need to grant access to additional devices that you own, you can add your public keys to your group's ~/.ssh/authorized_keys file, but note that this won't work until after the sign-up for this lab closes, because manually added keys could be overwritten by our auto sign-up scripts.
- You may wish to backup or write your code elsewhere. We suggest using SCP or SFTP to access your files. For Windows, WinSCP is a great tool. SCP and SFTP run on top of SSH, so use your SSH parameters (port, key, etc.) to connect.
- There's lots of online documentation for GDB. Here's one you might start with: GDB Notes (formerly hosted at CMU)
- Some "crash course in x86 and gdb" slides: section_lecture.pdf

Credits

This project was originally designed for Dan Boneh and John Mitchell's CS155 course at Stanford, and was then also extended by Hovav Shacham at UCSD. Thanks Dan, John, and Hovav!