# CSE 360 Project I

March 1, 2017

## 1 Change Notes

*Changes since version posted Mar 1:*

- Some advanced problems have been eliminated, while a few others have been turned into extra credit work.
- Part 3 of stack smashing exercise has been changed to remove uncertainty in stack offsets due to the use of `wrauth`.
- A few small changes were made to `my_malloc.c` in order to ensure that the vulnerability is exploitable.
- A few "bug fixes" were made to `vuln.c`.

These changes will likely invalidate any offsets you had previously computed, so you should recompute them. If you structured your code based on `driver_authd_expl.c` then this should be easy — you just run `vuln` inside `gdb`, print out the offsets again, and you are good to go. Fortunately, this is the final version of the vulnerable program, so this step will not have to be repeated.

Changes since the version posted Feb 28:

- Your group id is fully factored into the code. The `Makefile` uses the environment variable `GRP_ID`, which you should set before trying to compile any thing. You set the environment variable on the `bash` command line as follows:

  ```
  export GRP_ID=10
  ```

- Since the exploits are different with different groups, I can make a fully working sample exploit for the data-only exploit that overwrites `authd`. This exploit works when you set `GRP_ID` 10. This example will give you a road map on how to construct your exploit code, and how to structure it. (You will of course need to submit exploits that work for your assigned group id.)
- Additional minor modifications to provide more informative status and progress messages.
- The driver program now gives you an opportunity to attach to `vuln` process and debug it.

A (possibly incomplete) list of changes since the initial version:

- A driver program (`driver.c`) is provided that spawns `vuln` as a subprocess, and communicates with it. See the `main` function of `driver.c` to understand how this is done. You will develop your exploit by modifying this program. *You should not modify any other* `.c`*,* `.h` *or* `Makefile` *given to you.*
- Exercises have been reordered so that the simpler ones appear first, and the harder ones appear later.

- The Makefile now automatically generates an assembly code version of `vuln` in `vuln.s`. *To make the assembly file easier to understand, it now embeds source code lines within assembly,* so that you will know what line of source code results in which assembly instructions.

- Some minor simplification, modification or rearrangement of code have been made to `vuln.c`. For instance, a new function called `main_loop` was created because the stack frame for `main` is special and has some complications as compared to other functions. Now, you no longer need to worry about the stack frame of `main`.

  In addition, print statements have been cleaned up so that you can more easily see what information is being received by `vuln`, as well as the operations made by `driver`.

## 2 Overview

This project will be completed by groups of 3 students each, but in rare cases, groups may be permitted to have 2 or 4 members. Working in groups helps you learn from each other. Moreover, it allows the work to be split up, so that different exploits may be developed in parallel by different students. Note that there are three parts to this assignment. The effort needed is comparable across the parts, but stack smashing exercise certainly the longest.

You are given a vulnerable program `vuln.c` and a vulnerable heap implementation `my_malloc.c`. These programs, together with a Makefile, are provided as a tar-gzipped archive. Note that the program and the vulnerabilities are based on the problems in Homework 2.

Note that `vuln` accepts commands on its input and executes them. Examine the source code to see what the commands are. Note that it uses `read` rather than `scanf` or `gets`. This means you can input arbitrary values as input, a capability you need if you want to input arbitrary binary data that may include code or pointer values.

There are three basic vulnerabilities that you can exploit; for full credit, you will need to develop exploits for each of them. The vulnerabilities are:

- a format string vulnerability in `main_loop`,
- a heap overflow vulnerability in the version of `malloc` defined in `my_malloc.c` and used in `vuln.c`,
- a stack overflow vulnerability in `auth`.

The exploits you need to implement are described in more detail in separate sections below.

Note that you don't need to disable ASLR, stack protection or fool around with $W \oplus X$ to get your exploits to work. Instead, you will use the printf vulnerability to leak as much of the memory contents as you want. Initially, you will leak the contents of the stack. The stack will contain stack cookie — gcc uses the same value of the cookie for all functions, so you can read and reuse them. The stack will also contain saved base pointer. By reading it, you can overcome randomization of the stack base address. To cope with possible code randomization, you can read the return addresses off the stack. By dumping code memory, you can read information such as the address of functions in libraries (e.g., `bcopy`), and from there, you can compute the location of a more useful function such as `execl`. Finally, to overcome $W \oplus X$, note that the Makefile already makes the stack executable. In addition, `my_malloc` ensures that its heap blocks are executable.

## 3 Stack Smashing

Using the buffer overflow vulnerability in `auth`, implement the following:

- Use a data-only-attack on the local variable `authd`. In particular, use stack smashing in `auth` to go past the stack frame of `auth` into its caller's frame, and modify the value of `authd` there.

- Use a return-to-libc attack that returns to `ownme`. *Do not hard-code the address of* `ownme` *in your exploit.* Such a technique won't work if the base address of the executable is randomized. Instead, read the return address off the stack (using the format string vulnerability) and then compute the address of `ownme` from this information.
- A simple stack smashing attack that executes injected code on the stack that calls `ownme()`.
- (*Extra credit*) Use a return-to-libc attack that calls `execl` (or another function with a similar functionality) in `libc`, the standard C library. You should control the arguments so that you get a shell.
- (*Extra credit*) Use stack smashing to modify saved BP value on the stack frame of `auth` so that when control returns to `g`, you have control of the local variables of `g`, and can use this to set `s2` to `/bin/bash` even when `auth` returns 0.

# 4 Format String Attack

Implement an attack that uses only the format string vulnerability. Your goal is to execute arbitrary code injected by the attacker. (Your injected code can simply call `ownme()`.)

The format-string crafting strategy described in your homework solution is general, and will work across Windows and UNIX. However, since our target is Linux, which supports the `%n$` primitive, a simpler approach is available. In particular, you don't need the "skip" part of your exploit, but instead can start the "attack_format" part directly. You can perhaps set aside the first 128 bytes of your exploit string for this purpose. (128 bytes is much longer than necessary, but on the other hand, there does not seem to be much benefit in reducing the length.) The attack parameters can be in the next 128-byte block. This will be the data referenced by the `%n` directives in your exploit format string. Finally, you can put the injected code starting at offset 256 in your exploit buffer.

For this attack, you should obviously not overwrite the canary — you should selectively target the return address of `main_loop`, so that execution is diverted to the injected code when the quit command is sent to `vuln`, and it returns from `main_loop`.

# 5 Heap Overflow

Note that the heap overflow vulnerability resides within `heap_delete` function in `my_malloc.c`. This function is called from `my_malloc` as well as `my_free`. In theory, one could exploit it from either place. However, the heap blocks have to be arranged in a certain way in order for this work. So, you may need to use the `u`, `p` and `l` commands a few times to make sure that heap blocks are ordered in just the right way for your attack to work.

- Exercise a heap overflow in `my_free` to overwrite return address on the stack so that when `main_loop` returns, it executes `ownme`.
- (*Extra credit*) Exercise a heap overflow in `my_malloc` to overwrite return address on the stack so that when `main_loop` returns, it executes `ownme`.

To get this to work, you need to understand the implementation of `my_malloc` to a certain extent. In particular, you need to know the size of the blocks, the order in which the blocks occur in the free list, etc. You need to know the order because you can only overflow from a block starting at a lower address to a block beginning at a higher address.

Another challenge is that in `my_free`, there are two assignments:

```
current->next->prev = current->prev
```

and

```
current->prev->next = current->next
```

You can use the first statement to assign arbitrary value (contained in `current->prev`) to an arbitrary location (contained in `current->next`). Unfortunately, the second statement will interpret `current->prev` as an address and write to it. So, `current->prev` cannot point to the code segment. This means that your payload must be in writable memory, i.e., you need to execute injected code; it is not possible to do a return-to-existing-code attack.

You also need to figure out where your exploit code is going to reside. If you expect it to be in the heap block, then, keep in mind that the base of the heap managed by `my_malloc` is randomized, and will differ across runs. So, you need to figure out how to use the printf vulnerability to extract the base address. Alternatively, you can see if the exploit code can be put on the stack, whose addresses you have already figured out. (This is what I did.)

# 6 Submission

Your submission will be in the form of C-programs. In particular, for each exploit, you will create a version of `driver.c`. Compiling and running this exploit program should lead to a successful exploit. *Note that you need to submit the source code for the exploits.* You should not change `vuln.c` or any of the other material provided to you.

You should create a tar-gzipped archive of all your exploit programs. Give them descriptive names such as `driver-smash-data.c`, `driver-heap.c`, etc.

Submission will be on Blackboard, and the details will be provided to you.

# 7 Tips

- Use the 32-bit VM image provided to you. Your submission will be tested on this VM, so you might as well work on the same VM.

  I have tested the exploits on a 64-bit Ubuntu system, when every thing is compiled with the `-m32` flag that produces 32-bit binaries. However, before you submit, please ensure that your exploits work correctly on the provided VM.

- **Don't change the** `Makefile`, except possibly for adding additional lines for compiling additional exploit programs.

- **Review carefully the example exploit program** `driver_authd_exp.c`. You will gain a better understanding of how to structure your exploits, and also save time on other exploits.

- You can print a specific offset that is, say, 100 words from the top of the stack using `printf("%100$x")` instead of having to use 100 instances of `%x`'s. (Note that this may end up printing something that is a few words off, say, 97 words from the top of the stack.)

- Within gdb, registers can be accessed by prefixing them with `$`, e.g., `print $esp` will print the stack pointer register.

- Within gdb, you can print arbitrary memory locations by casting them into pointers and dereferencing them, e.g., `print *(int *)0xbfffff7c`. You can control the format, e.g., print it in hex using `print /x *(int *)0xbfffff7c`.

- You need to use the printf vulnerability to leak several pieces of information. The first is the stack canary value. The second is the saved ebp value that you need in order to figure out the base of the stack frames. (You cannot hard-code stack base address because the stack base is (re)randomized on each execution.) Finally, you need to leak return address on the stack, or the address of library functions in the GOT (Global Offset Table).

The driver program is necessary because of the need to leak these pieces information. You will structure your exploits as follows. First, you will use the `e` command to leak the above pieces of information. You will extract the information into variables in the `driver` program, which will then construct an exploit string and send it to `vuln`.

- You can debug an already running process by using `gdb` to attach to it. (Some times you may need `root` privilege to attach to an existing process.) To attach to an existing process, e.g., `vuln`, type `ps ax|grep vuln` at the bash command prompt. It will produce a list of processes that have the name `vuln`. Note down the pid, fire up gdb, and at its command line, type `attach` to that pid.

  *This ability is invaluable for tracking down problems with your exploits.*

- If you want to do the extra-credit problems, then first use `objdump` to disassemble the executable. *An executable contains code that won't be in the object file* `vuln.o`*, or the assembly filr* `vuln.s`. Use `objdump -d vuln` to disassemble the executable. Then you will see how library calls are made, and how you can hijack them.

Although the stack and code layout is going to be different for each team, the layout does not change from one run to another. So you can use gdb to figure out the layout once, and then use it repeatedly in your exploits. Specifically, you need to know the size of the stack frames of `main_loop` and `auth`, and you can find this by running `vuln` within gdb, setting break points in these functions, and printing the values of `ebp` and `esp` registers. Make sure that you print `esp` value after the calls to `alloca`. (This function allocates storage on the stack, and hence will change the value of `esp`.)

*In order to succeed in this project, you have to get good at using gdb if you are not already there.*

### 7.1 Working with assembly/object code

Some exploits require you to use binary code. You can do this by writing a small assembly code snippet and then compiling it using an assembler. One option is to use `as`, the default assembler on your system. You can invoke it as:

```
as -a --32 test.s
```

where `test.s` is the file containing your assembly code. This command dumps the assembled code on the screen. Note that `as` uses AT&T syntax for assembly. Alternatively, you can `nasm` which supports Intel format. (I have not used nasm.)

Instead of trying to use direct jumps or calls to absolute memory locations, you should try to use indirect jumps and indirect calls. First move the target address into a register, and then use an indirect jump or call using that register. Various other points to note:

- Make sure you get your assembly syntax right for various addressing modes and operands. Specifically, for `as`, make sure you prefix immediate operands with a `$`, and register operands with a `%`. For instance, `mov $0x20, %eax` moves the decimal number 32 into the register `eax`, while `mov 0x20, %eax` moves *the contents of memory location* `0x20` into `eax`. Also make sure that you use a `*` for indirect calls and jumps, e.g., `call *%eax` is an indirect call to the address contained in `eax`. (However, `call *(%eax)` first dereferences the location whose address is in `eax`, and then fetches the value stored at this memory location, and then calls that location.)

- You can use `gdb` to work at the assembly level. You can use `layout asm` to see your code in assembly. You can use `stepi` to single-step assembly instructions. For more details, see:

  `https://www.csee.umbc.edu/~cpatel2/links/310/nasm/gdb_help.shtml`

  For even more details, see:

  `https://sourceware.org/gdb/onlinedocs/gdb/Registers.html`

  Another useful link I found was:

  `https://cs.nyu.edu/courses/fall03/V22.0201-003/c_att_syntax.html`