Computer Security - Semester Review

Guilherme Gomes Haetinger

University of California, Berkeley
Fall 2019
- Share this with whomever you want. If you spot a mistake, email me at ghaetinger@gmail.com -

Contents

1 Security Design

Let's discuss on how to make a system theoretically secure, as in the decisions that should be made and what are the main points we should leverage when designing a system from scratch.

1.1 Security Principles

We have a number of security principles that must be considered once we are developing software [1]. Some of them can be enumerated as the most important. These are:

- **Security is Economics**: Only spend as much money as what you are trying protect is worth. Don't buy a $10000 lock for a $10 bike.
- **Least Privilege**: Only give a program the actual amount of privilege it needs to do its purpose. We should not give ROOT access to a program that plays the nyan cat video.

- **Fail-safe Defaults**: Safe defaults in the sense of "*if something fails, what should be the current state?*". It's recommended that we use *default-deny* policies. The light goes down on a server's building. Should the electronic lock on the server access door be unlocked or stay locked? $\rightarrow$ If we want *default-deny* policies, the door will stay shut, keeping the server's hardware safe from whomever wants to access it, which will keep it secure from someone who jammed the building's circuits just to get to the computers.
- **Separation of Responsibility**: Separate privilege. "Nobody has full privilege by itself". Nuke triggers need multiple people turning a key to work (at least in movies).
- **Defense in Depth**: Create redundant layers of protection. In the medieval times, castles were protected by an outer wall and an inner wall, so that enemies would have to go through 2 different walls to truly invade it. This made the process much harder.
- **Psychological Acceptability**: "Users must buy into your security model". If you want users to use your safety resources, make it easy to do so. If to process a company transaction, the user is asked to fill a form of 100 pages, after processing a number of transactions, the user will tire and just leave the form aside, hoping that nobody checks it.
- **Human Factors**: Always consider human factors. Things must be usable. Don't make it hard for an ordinary user to interact with your system. Don't make regular users think of a password with 15 different upper case letters, all the letters in the alphabet and at least 5 letters of the ancient Greek alphabet.
- **Complete Mediation**: Make sure you have control over **every** point of access. Enforce access control policies. Bottleneck the airport's immigration procedural check so you know every immigrant is treated the same controlled way.
- **Know your threat model**: Consider changes in your threat model. Keep track of it and ensure you are safe from it. Internet was made for researches with no threat model whatsoever. When they opened for the public, they had to consider the malicious use of the internet. It now had a threat model.
- **Don't rely on Security through Obscurity**: Don't rely on the fact that your design/algorithm is secret. You design a code that sends the user's password unencrypted back to the server for some weird reason. You provide your system as compiled code, in a way that the user can't interpret it correctly. The user may be able to reverse engineer it and hijack the server connection to get other user's passwords.
- **Design Security from the start**: Don't leave security out for refactoring. It's usually really difficult to refactor code in order to make it secure because it needs a system redesign. A webpage has users and their given passwords. They allowed whatever password the user wanted to use on it. They decide to restrain to safer 16-character passwords with all the right shenanigans. What happens to the already created accounts? Do you make them redo their password, which should take a lot of time? Or do you maintain them unsafe?.
- **Kerkchoff's Principle**: Similar to the *Don't rely on Security through Obscurity* principle, this one asks you to "Design your system as if the attacker could read your code". We can consider the same example as the referred principle.

1.2 Security Design Patterns & What you should think about when designing your system

How should we go at *Developing your own secure system*?

1.2.1 Trusted Computing Base - the TCB

The simpler definition for this abstraction would be *the part of the system in which we rely so that it works properly*, meaning that no problem outside of this part can obliterate your service. Now, the point of this design is to minimize it so that it is easier for us to place our trust in it. It's easier to place your trust in a 10 line code than in a 100.000 line code. We want the **TCB** to be *unbypassable*, *tamper-resistant*, *verifiable*. It is called a *primitive yet effective kind of modularity*.

1.2.2 Modularity and Isolation

The whole idea of modularity can impact on different levels of the system. Sometimes it can bring efficiency by assuming that each module already has it's own required data to run correctly. It can also provide legibility since we can understand how the code is divided into multiple responsibilities, which can impact on refactoring (can eventually help quick security patches). Now, most importantly, modularity can provide us with *isolation*, meaning that each module is independent and can keep its problems to itself → minimizes assumptions made by other components that it interacts with, enabling them to treat errors in the system without crashing it (understand what happened to the other component and act accordingly).

2 System Implementation Vulnerabilities

What are the main system threats regarding its code implementation? Let's see how we can generally exploit and fix these vulnerabilities.

2.1 Time-of-Check To Time-of-Use (TOCTTOU)

This is a general vulnerability (when I write general I mean it can happen in any logical programming environment (when I say environment I mostly mean language (when I say language it's just because Weaver specifically tells us not to use C))). As the title already says, this vulnerability takes into account the time of check for a variable and the time you assign its value. Take a look at the following code:

```
def openFileOfSize200(size, filename)
  if metadata(filename).size > 200
    print "Haha this is unbypassable"
    exit
  end
  # Sleep a bit because there is definitely another process that needs CPU more than I
  sleep(1000)
  read(filename, 'r')
end
```

This code has a flaw. As you can see, its purpose is to only read files that have the size less or equal to 200. The code reads the file metadata and checks it size. If it's bigger than the purposed value, it exits. What if I changed the file size while the program sleeps? → The file with the larger size is read in the end, because the time of check, which is when the if-statement is run, for being far away from the time of use, enables us to bypass the check.

2.2 The Stack & How C breaks it (Memory Safety)

Before you read anything from this section, take a look at the Appendix section on Assembly code! There is a lot of review on it needed for this part of the content. Now that that's out of the way, let's smash the stack.

2.2.1 Format String Vulnerability

For this exploit, it **very** important to understand the layout of variables inside the stack. For this, see the appendix notes on it. We're all very familiar with `printf`. It can take *1 to n* arguments, being the first a string with **hotkeys** such as `%d, %c, %f, %s, ....` These keys represent the format of representation of a given argument. If someone just prints out user input with `printf`, the formatting string (the one with hotkeys) will be determined by the user, meaning that it can use whatever formatting string. What can a user do with its arbitrary formatting string, when the number of arguments given to printf is smaller than the number of hotkeys?. Considering the structure of the stack when `printf` is called, the hotkeys will make the function look for a specific argument that doesn't exist, which will make it interpret whatever is in the Stack in argument's position as the one itself. The following example might clear up what I'm passing on:

```c
int main() {
  int num = 100;
  char buf[10];
  if(fgets(buf, sizeof buf, stdin) == NULL) return 0;
  printf(buf);
}
```

If we use the input `%s%d`, we'll get the value of `buf` followed by the value of `num`. This happens because the argument that we seek to fill `%s` will be the first memory slot above the formatting string argument and, since there are no other arguments, it will fall on the local variables of the `main` function. Hence, `%d` will take the value of `num`, which was declared right above `buf`. Now what would we use this for? Maybe getting the internal state of the program might be interesting for your exploit (emphasis on **Stack Canaries**).

There is another way to approach this exploit by using a specific hotkey that enables you to write the value of printed characters (until it's called) in some memory address. This hotkey is `%n`. We can do something like this to exploit the same code but with `buf` declared before `num`. Given a number *z*, we can store *z* in an arbitrary address *a* by inputting the following string: `a%(z-4)x%n`. The `printf` function will print the 4-byte address, followed by a (z-4)-byte word format of `num`, which is the last pushed local variable, and, finally, will read the first 4 bytes of `buf`, which happen to be *a*, and use it as input for `%n`, storing $z - 4 + 4$ in *a*.

This vulnerability is easily fixed by calling `printf("%s", buf)` instead of `printf(buf)`.

2.2.2 Integer Conversion & Overflow Vulnerabilities

This is a simple vulnerability. Always check the type of your input as you use it in other functions. Be careful because negative `int` values can be less than whatever size check you have in your code but be extremely big when converted to `unsigned` types that are used in standard writing functions such as `memcpy`.

Also, be careful when using arithmetic operations when trying to allocate the correct amount of space for a variable. Values can overflow and allocate a much smaller memory chunk for that variable, allowing a sizable input to overflow your small sized buffer.

2.2.3 General Protection Against Memory Attacks

- Secure code Practices
  - Check validity of variables (not `NULL`, within bounds, ...)
  - Use standard safe functions such as `strlcpy` instead of `strcpy` and `fgets` instead of `gets`
- Using a memory-safe language
- Runtime checking
- Compiler's static analysis
- Testing
  - Test generation, Bug detection
  - Random, mutated and *structure-driven* inputs.

2.2.4 Buffer Overflow

This is the easiest vulnerability we were able to exploit in this class. As a trade-off of being easy to exploit, it is also easy to fix.

Given a program in a language that doesn't implement memory safety (C), we can have programs that for a given input behave maliciously. We can do this via the *Buffer Overflow* vulnerability in some programs. This is, nonetheless, the ability of filling a variable with a value that doesn't fit in it, enabling us to write on the memory that is above it in the Stack. For example:

```c
int main() {
  char input[4];
  gets(input);
  return 0;
}
```

We know that `gets` reads whatever you input and writes it into a variable with a `'\0'` in the end. What happens if we input the output of the following python code in it?

```python
print("a"*4 + "b"*4 + address_for_malicious_code)
```

What happens is (considering no callee registers):

- The variable `input` will have been filled up by `"a"`s;
- The `EBP` value will have been filled up by `"b"`s;
- The *return address* will have the value of the address pointing to a malicious code (We probably should input the malicious code as well, but that would involve calculating the actual address of the variable `input`).

In the end, our stack would have the following layout:

LOWEST Mem Addr.	ESP	4 bytes
"input"	4 bytes	
EBP	4 bytes	
Return Address	4 bytes	
...		
HIGHEST Mem Addr.		

We can also use this to change variables that are on top of the input variables.

2.2.5 Stack Smashing Mitigation

This is a more dense subject. Considering that buffer overflow is one of the most common exploits, the following mitigation options are more complex and are harder to barge through.

1. Stack Canaries

3 Appendix

3.1 Assembly code for Immediate suffering

Let's review some topics for the Assembly code structure when generated through C code.

3.1.1 Registers

For the purpose of this class, I'm sure we'll only need to know 32-bit registers (not that there are many differences between 32 to 64, but the names differ).

- Data registers [2]:
  - **EAX** → Accumulator: IO and Arithmetic functions, **Is where the return value is stored**;
  - **EBX** → Base: Indexed addressing;
  - **ECX** → Count: Loops
  - **EDX** → Data: Basically the same as **EAX**;

- Pointer registers
  - **EBP** → Base: Holds the base address for the stack;
  - **ESP** → Stack: Holds the top address for the stack;
  - **EIP** → Index/Instruction: Holds the offset for the next instruction

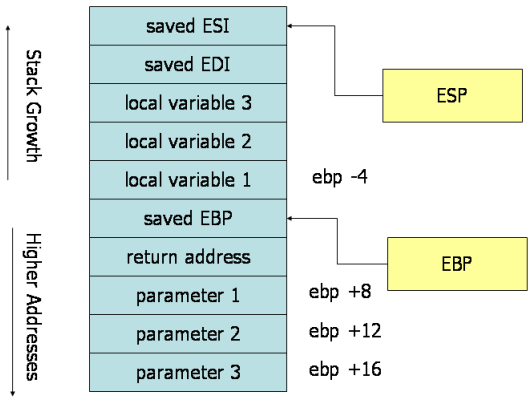

Figure 1: The Stack registers layout

3.1.2 How do function calls work?

This part is really important for us so we actually understand how the stack layouts itself on *return* and exploit the return address. It follows these operations [3]:

- Setup & execution
  - Push all the function parameters into the stack (piles up from last to first → first one in the lowest memory address);
  - Call the function by running `call`;
  - Push the *Return Address* into the stack;
  - Points `EIP` to the start of the function;
  - Save the previous `EBP` on top of the stack;
  - Set `EBP` and `ESP` to point to the value of the old `EBP` (top of the stack, which means `ESP` was already pointing at it);
  - Stack the `callee registers`;
  - As the local variables are declared, we decrease the value of `ESP` to increase the size of the stack frame;
- Return
  - Store the return value in `EAX`;
  - Pop the `callee registers`;
  - Make `ESP` equal `EBP`;
  - Pop the old `EBP` to `EBP` (`pop ebp`, `ESP` will increase value because the stack size gets smaller);
  - As `ESP` now points to the *return address* (which was stored right on top of `EBP`), `ret` will make the `EIP` point to the correct address.

3.2 Variable Layout in the Stack

We have some important fields and their data size. Their data size is the amount of space they occupy in the Stack, Heap, etc. These are:

int	4 bytes
float	4 bytes
double	8 bytes
char	1 byte

Consider that `long, short` usually increase and decrease, respectively, around `4 bytes`.

For structures, however, we have a more complex layout. The first declared variable will be in the lowest memory position and the last one in the highest. The following example shows the Stack layout once we declare a structure variable:

```
typedef struct {
  int i;
  char c;
  float f;
  double d;
} Sample;

int main() {
  Sample sample = {1, '2', 3.0, 4.0};
  return 0;
}
```

Given this code, once we execute `main`, we'll have the following structure in the Stack:

LOWEST Memory Addr.	ESP	4 bytes
	sample.i	4 bytes
	sample.c	1 byte
	sample.f	4 bytes
	sample.d	4 bytes
	EBP	4 bytes
HIGHEST Memory Addr.	return address	4 bytes

References

[1] David Wagner. Security principles notes, 2019. `https://cs161.org/assets/notes/Principles.1.19.pdf`.

[2] David Evans, 2018. `https://www.cs.virginia.edu/~evans/cs216/guides/x86.html`.

[3] Zeyuan Hu, 2017. `https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/`.