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:

```
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
  - Preconditions and Post-conditions
- 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:

```
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?

```
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**

   Random value generated when program starts that is stored below the `EBP`. Its value is checked once the function returns and, if it has changed, the program will know it has been hijacked. The idea behind it is to avoid simple Buffer Overflows to change the value of the `EBP` or the *return address*. How can we go around this mitigation? We have to find a way *not to kill the Canary*. For this, we have the following options :

   - Find out the value of the canary and rewrite it in the process of modifying the `EBP` or *return address*. To do that we either have to find a string formatting vulnerability that may print the value or any other information leak that might dump it, e.g. finding a way for the program not to read a `'\0'` character in the end of a string while printing it out will make the program leak every information until the next `'\0'`. The example can be easily prevented by making the first bytes of the canary always be equivalent to the end-string character, making it stop before reading the canary. While this is effective against this attack, we can see that the entropy of the canary is lowered by 25%, which is supposed to make brute-force plausible since we now have 24 bits of entropy.

- Use a string formatting vulnerability to write around it with specific addresses.

2. **Non-executable pages**

   We maintain the permission of writing and executing in a `XOR` condition, meaning that the program can either write code on stack/heap or execute it. This is insufficient since it is easily breakable by **Return Oriented Programming**, which is basically changing the return addresses of the code to known portions of standard libraries or even the code itself, i.e. using already written code as modules for writing your own malicious execution.

3. **Address Space Layout Randomization**

   Consists on rearranging/relocating the chunks of memory into different addresses. This makes things much harder to exploit considering that we don't have fixed addresses to write code and then redirect the execution to it. Together with **Non-executable pages**, requires an information leak to be broken. How can we bypass this? There are some ways we can do it, but require really specific scenarios [2] and are probably not worth getting into.

3 Cryptography

Cryptography is the field of studies and implementations regarding algorithms and systems that seek to ensure **Confidentiality** (Prevent others from reading our data without authorization), **Integrity** (Prevent others from modifying our data without authorization) and **Authentication** (Asserting the identity of someone who sent a message, edited a file, etc.) to our private data. Weaver clearly emphasizes the fact that these systems are not meant to be redesigned by us because they are really easy to screw up. We'll keep using message sending as our example for every algorithm.

3.1 Independence under Chosen Plain-Text Attack Game (IND-CPA)

The IND-CPA game is designed to check whether the algorithm in question is not deterministic and, thus, is a step closer to safety. It consists of the following steps:

- Attacker sends two messages to an *Oracle* (Entity that encrypts the messages using the encryption algorithm being tested and the *K* key unknown to the Attacker);
- *Oracle* replies the encryption of one of them;
- Attacker can do this for any message it wants as many times as it needs;
- If Attacker has, at any point, more than 50% chance of guessing which message was encrypted, it wins the game.

If the Attacker wins the game, we can assume that the output of the encryption algorithm can be predicted by its input even without knowing *K*. This makes the algorithm extremely flawed because someone eavesdropping an encrypted conversation might be able to deterministically understand and hijack it. Therefore we seek the algorithms that win it.

3.2 Symmetric Encryption

Given that two people (A, B) have a secret key *K* known only by them. Consider $E_K(M)$ as the encryption function and $D_K(C)$ as the Decryption function (M is the message and C is the cypher).[3, **?**, **?**]

3.2.1 One Time Pad

This is the most simple encryption algorithm we saw. The calculations are self-explaining.

$$C = E_K(M) = M \oplus K$$
$$M = D_K(C) = C \oplus K$$

It's trivial to understand why this is a IND-CPA loser. This example shows why

$$A \rightarrow_{M1,M2} O \tag{1}$$
$$O : C1 = M2 \oplus K \tag{2}$$
$$O \rightarrow_{C1} A \tag{3}$$
$$A \rightarrow_{M2,M3} O \tag{4}$$
$$O : C2 = M2 \oplus K \tag{5}$$
$$O \rightarrow {}_{C2} A \tag{6}$$
$$A : C1 = C2 \rightarrow C1 = C2 = M2 \oplus K \tag{7}$$

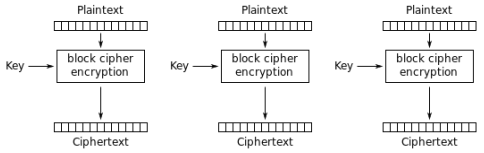
Note that (2) and (5) are random, meaning that the oracle could have chose $M1$ on (2) or $M3$ on (5). However, this is irrelevant since the Attacker can just repeat the operation to get the only two possible outputs (it's exactly two because the "$\oplus$" operation is deterministic) and then compare which output repeats on both cases. Note that once it knows the encrypted message, it can simple derive the key *K* from $K = C1 \oplus M2$.

3.2.2 Block Ciphers

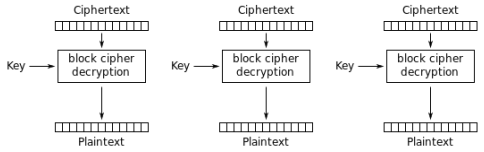
Block Ciphers divide the message M into blocks and encrypt each one with *K*. The encryption algorithm itself is deterministic. The ones I'm listing are the most important ones. CFB has the same properties as CBC buck it's worse with IV reuse. [4, 3]

1. **Electronic Code Book**

   The simplest block cipher. Every chunk of code goes through the same encryption process. It can be defined by the following: $C_i = E_K(M_i), M_i \in M$. The encryption is deterministic and the ECB doesn't do anything to keep the Ciphers from having no entropy whatsoever. This means that chunks with the same value in the message will have the same cipher output, leaking information and, clearly, losing the IND-CPA game.



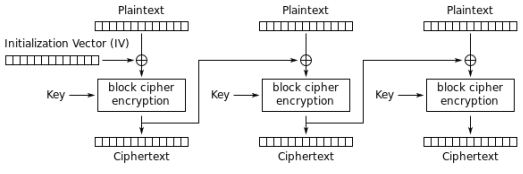
Electronic Codebook (ECB) mode encryption



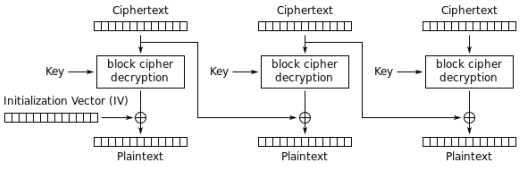
Electronic Codebook (ECB) mode decryption

2. **Cipher Block Chaining** This block cipher uses the concept of nonces (*nonsense*) to add entropy to the cipher output. If the IV (nonce) isn't reused, CBC is IND-CPA. Note that although we can't parallelize the encryption, we can parallelize the decryption. The structure is the following:

$$C_0 = IV$$
$$C_1 = E_k(C_{i-1} \oplus P_i)$$
$$P_i = D_k(C_i) \oplus C_{i-1}$$

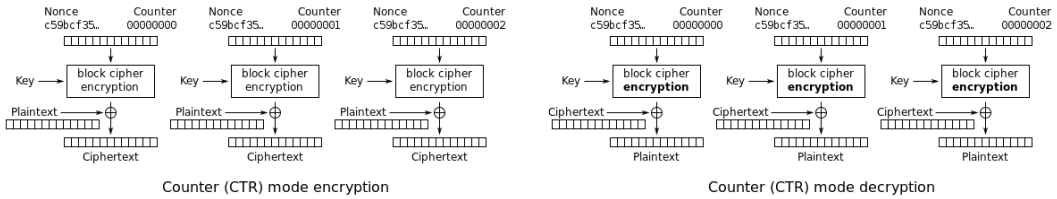


Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

3. **Counter Mode** All-parallelizable alternative to CBC. Instead of one IV, uses $nonce||i, i \in [0, ..., n]$ with *n* being the number of blocks. Reusing the IV in CBC makes it lose the IND-CPA game, but it's not catastrophic because the attacker will only find out the first block. If you reuse the IV in CTR mode, it transforms itself into the one time pad, leaking all the message. Note that CTR decryption uses the nonce encryption instead of a decryption function and that it is not exactly a block cypher but more of a stream cypher because it doesn't need padding. The structure is the following:



3.3 Hashes

Hash functions have the following:

- Variable input size
- Fixed output size
- Efficient computation
- Pseudo-Random

Along with that, we must make sure that the hash functions provide us with these properties:

- One-way: easy to compute, almost impossible to revert the process.
- Second Preimage resistant: given *x*, almost impossible to find a *x'* s.t. Hash(x) = Hash(x').
- Collision resistant: Almost impossible to find x, y s.t. Hash(x) = Hash(y) (Collision resistance implies second preimage resistance)

3.4 Message Authentication Codes

MACs are checksums for messages, meaning they are sent with the message so that they are proven to be untampered. MACs, as encryption algorithms, need a key to be private only between the ones that are sharing the message. As an example:

$$
\begin{aligned}
&A : C = E_{EK}(M) \\
&A : T = MAC(MK, C) \\
&A \rightarrow_{\{C,T\}} B \\
&B : T = MAC(MK, C)?
\end{aligned}
$$

If the operation on the last line says that they are not equal, B can assume that the message has been tampered with. Also, the convention of using the MAC in the encrypted message is the correct way of using it since **MACs are deterministic and leak information**. **Never use the same key for MAC and Encryption**.

3.5 Password Storage

Passwords should not be stored in a map `\{User, Password\}` because if someone gets ahold of this structure, they can simply impersonate everyone. We want the password to be a secret to our server. To do that we can store the map `\{User, Hash(Password)\}`. This way, if someone gets this information, they can't do much because of the *One-way property* of hashes. The user would input the password and send the hash of it to the server, which would then check if it's correct.

However, many people choose poor passwords. Consider that the attacker knows what's the hash function by using the server as an oracle. What can he do with the fact that there are simple passwords scattered around many users? He can create a brute forced dictionary with all the simple passwords and their hashes so that, when it gets ahold of the server's structure, it can just match the hashes with its dictionary (Dictionary Attack). We can mitigate that by adding a random, user-specific (unique) Salt (giant random number) to the user's password before it's hashed. This way, even if the attacker can get its hands on the hash, they will never match the weak password's. We also would like for the hashing process to be slow, which would make the brute forcing much slower.

3.6 Pseudo Random Number Generators (PRNGs)

A PRNG needs the following functions: `seed(x)`, `reseed(x)`, `generate(size, x=optional)`. The *x* in the functions are the true sources of entropy. The PRNGs should be predictable only if you know its internal state and should be roll-back resistant, meaning their internal state at time *T-1* should not be determinable by someone with the knowledge of the state at *T*.

3.7 Asymmetric Encryption

Although symmetric encryption looked as if it had everything taken care of, we must understand how *A* and *B* shared the secret key *K*. If they haven't met up in person to share it, it is very likely the key is not a secret. Let's look at some encryption algorithms that take care of this aspect.

3.7.1 Diffie-Hellman

The event *A* wants to communicate with *B* follows this sequence of operations for *p* (huge prime) and *g*, $1 < g < p-1$ public.

$$
\begin{aligned}
&A : a \in \{0, 1, ..., p-2\} \\
&\quad : X_A = g^a \mod (p) \\
&B : b \in \{0, 1, ..., p-2\} \\
&\quad : X_B = g^b \mod (p) \\
&A : broadcast(X_A) \\
&B : broadcast(X_B) \\
&A : S = B^a \mod (p) = g^{ab} \mod (p) \\
&B : S = A^b \mod (p) = g^{ab} \mod (p)
\end{aligned}
$$

Now that they have a symmetric key *S*, they can simply use a symmetric encryption system. Have in mind that a man in the middle can read g^a, g^b and g^{a+b} but can never get to g^{ab} because they don't know *a* or *b* and can't break the modular equation because that would require it to break the discrete log, which is extremely hard. However, a man in the middle can intercept the messages and just create its own X_M and *m* to structure two different channels: one with *A* and one with *B*.

3.7.2 El-gamal Encryption

This algorithm uses a similar idea as *Diffie-Hellman's*. We have all the information listed above. From that, we want *A* to send a message to *B* (the message must be $\in \{1, ..., p-1\}$). We do:

$$
\begin{aligned}
&A : r \in 0, ..., p-2 \\
&A : \{R, S\} = \{g^r \mod (p), M * X_B^r \mod (p)\} \\
&A \rightarrow_C B \\
&B : R^{-b} * S \equiv g^{-br} * M * X_B^r \equiv g^{-br} * M * g^{br} \equiv M \mod (p)
\end{aligned}
$$

We know that only *B* can decrypt it because it is the only one that knows its private key *b* and can compute R^{-b}. We must not reuse *r* or else it will leak information.

3.7.3 RSA Encryption

This one is definitely the most complicated one. *A* generates two private primes *p, g*; calculates public $n = p * g$ and private $\phi(n) = (p-1)(g-1)$; picks random public $2 < e < \phi(n)$; solves private $d = e^{-1} \mod (\phi(n))$. Now if *B* wants to send a message to *A*, this is how it goes:

$$
\begin{aligned}
&B : C = M^e \mod (n) \\
&B \rightarrow_C A \\
&A : C^d \equiv M^{ed} \mod (n) \\
&\quad : ed \equiv 1 \mod (\phi(n)) \rightarrow ed - 1 = k\phi(n) \\
&\quad : M^e d \equiv (M^{\phi(n)})^k * M \equiv_{Euler_{thm}} 1^k * M \equiv M \mod (n)
\end{aligned}
$$

However, the encryption is deterministic for the same *e*, meaning it can leak information! There should be a source of entropy in it, but it isn't that important for our scope.

3.8 DSA Signatures

Based on Diffie-Hellman. Signatures are used to avoid the man in the middle attack listed in 3.7.1. Initial parameters: `L, N, Hash(x)`. *A* creates an N-bit prime *q*, L-bit prime *p*, s.t. $q|p$ and $g = h^{\frac{p-1}{q}} \mod (p), 1<h<p-1$. We say *p,g,q* are public. *A* has the private key *a* and the public key $g^a \mod (p)$. *A* calculates a random $k < q$. And the process of signing by *A* and verifying by *B* goes as such:

$$
\begin{aligned}
&A : r = (g^k \mod (p)) \mod (q), r \neq 0 \\
&\quad : s = k^{-1}(Hash(M) + ar) \mod (q) \\
&\quad : Sign = \{r, s\} \\
&\quad \rightarrow_{...||Sign} B \\
&B : w = s^{-}1 \mod (q) \\
&\quad : u_1 = Hash(M) * w \mod (q) \\
&\quad : u_2 = rw \mod (q) \\
&\quad : v = (g^{u_1} g^{au_2} \mod (p)) \mod (q)
\end{aligned}
$$

And if $v = r$, the signature is valid!

3.9 Certification and Authentication

We trust a certificate authority, which gives us the correct certificate (public key) for whomever we want to speak to. The certificate should be signed by the CA itself or by someone the CA has signed a certificate to, meaning that it is trustworthy. The latter is the method for certification scaling, avoiding the bottleneck of having one CA authenticating numerous users and implementing a signature delegation (Hierarchical). We should also be able to receive a list of certificate revokations so we can avoid insecure connections.

4 Web Security

This section will focus on the possible exploits and mitigation options for web pages as well as the structure of the web.

4.1 SQL Injection

This is probably the most known system attack these days. It's really simple to understand and avoid.

Web servers/pages can have databases. The most known Database Query language is SQL. Usually, the web server/page would have queries to fill out information on the HTML rendering and such. For this, they must, sometimes, read user input and add it as a field to the query code. You have a social media website. You want to display someone's profile when it is searched for. You create a field in the web page asking "Looking for...?". The intended output of a query in this scenario would be all the users information (name, age, ...). So you add the following to your code:

```
SELECT name, age, photo FROM users WHERE username = input;
```

What can an attacker do? It can use a string like the following: `"; DROP TABLE users;--`, which will end the query, break the line, add a *drop table* command and comment whatever is in the right just to make sure. This will break the website. The best mitigation for this are *prepared statements*, which will send the input all the way to a leaf inside the tree structured query, taking away all its privileges of running any commands.

4.2 Javascript & Cookies

Javascript is a scripting lenaguage used to structure and add functionalities to your webpage. It can basically do whatever it wants to the **DOM** as well as read/change the session cookies. In order to make your computer safe from dangerous websites with Javascript that you may end up visiting, Javascript is *sandboxed*, which means it cannot change files or programs in the computer it is being run on.

4.2.1 Same-Origin Policy

Another danger of Javascript is the fact that it should be able to tamper with other websites. That's why there is **SOP**. The policy restrains webpages from affect things that are out of their origin. For two origins to be the same, they must have the same `domain, port, protocol`. For example, the website `http://amazon.com/idk/1023` has the same origin as `http://amazon.com:80/babyyoda/1123421` but not of `https://amazon.com/idk/1023`, `http://store.amazon.com/idk/1023` or `http://amazon.com:1234/idk/1023`.

4.2.2 Cookies!

Cookies are basically variables for someone's session in a website. They are structured with a `host, path` key and a value. This is how companies track what Ads you want to see. Here are some cookie properties:

1. **Cookie Origin Policy**

   Don't confuse this with SOP! Cookie origin policy restrains the cookie sharing process as in *"which websites can receive my cookies"*. The only websites that can share your cookie are the ones that have matching suffixes. E.g. example.com/foo shares with www.example.com/foo but not with example.com/bar.

2. **Secure Flag**

   A cookie can be set with a secure flag, which means it can only be sent via *HTTPS* connection.

3. **HTTP-Only Flag**

   A cookie set with the HTTP-Only flag will only be accessible in the server side.

4.2.3 Spectre Attack

A hardware side channel developed to access the nested iframe's information by treaking the processor's predictor. Solution is to make the browser **eat RAM** by giving every website it's own process.

4.3 Cross-site Forgery Attack (CSRF)

Attacker inserts malicious request in a page it has control inside an `<img>` tag or any other operation that causes the request to run. E.g. You visit the attacker's webpage, which files a transfer request in your name on www.bank.com. Is that going to work?. It depends on how the bank implements its system. The point of this attack is to take advantage of the cookies saved for the victim. So if the bank determines who is logged in by the cookies, then the attack might work. There are two important mitigation options for this exploit:

4.3.1 "Referer" validation

The "referer" validation allows the website to identify from which website the request is coming from. If the website is not itself, it makes sense to terminate the operation. However, there are still some ways of going around it.

4.3.2 Secret Token Validation

A short TTL random cookie that defines the login session for the user. We can add this cookie to the URL, making the URL itself hold entropy, which means the attacker's only option is to brute force the cookie (probably infeasible) or find a way to retrieve it. The solution, however, is not really pretty and Weaver doesn't consider any solutions regarding the STV perfect, but it is better than the "referer" validation.

4.4 Cross-site Scripting (XSS)

Its main objective is to Subvert the SOP.

4.4.1 Stored XSS

Persisting a malicious script (Javascript) in a server that's going to be accessed by the victim, running the code. In the end, it can either execute whatever malicious operation it wants in the website using the victims authentication cookie or it can send data (cookies mostly) to the evil server.

4.4.2 Reflected XSS

User enters malicious website, clicks on a link that redirects him to a website whose cookies are interesting with a script in the URL. The script can either send data to the attacker or perform malicious operations in the name of the user.

4.4.3 How to fix?

- Input sanitation (CSP) $\rightarrow$ Removes whatever might be fishy about an input
  - Prohibit inline scripts
  - Whitelisting
  - Blacklisting
- HTTP-only flag $\rightarrow$ doesn't allow the user to access its cookies, so the script can't get them.

4.5 Clickjacking

Placement of invisible i-frames of a target website over some enticing content or placement of visible of a target under a malicious invisible frame to capture user input. We can fix it by frame-busting: website prohibits its page from being a frame or by using the HTTP **X-Frame-Options**, which whitelists some websites for frame use.

5 Network Security

This is most certainly the hardest part of the content. This is an overview for Networking as well as an observation set for its vulnerabilities.

5.1 The 5 layers that actually matter

We can divide networking into 5 different layers, the lowest their number, the lower levelled they are. These are:

7	Application	Communication of whatever (e-mail, torrent,...)
4	Transport	End-to-end communication between processes (TCP/UDP)
3	(inter)Network	Bridges multiple "subnets" to provide end-to-end connectivity
2	Link	Framing and transmission of bits into a message
1	Physical	Encoding bits to send them over a link (single)

5.2 Basic stuff

- IPv4 → 32-bits
- IPv6 → 64-bits
- localhost → 127.0.0/24
- Broadcast → 255.255.255.255
- Protocols → agreement on how to communicate (Format, order of message, etc.)
- Packets → What communication is broke into
  - Software doesn't see it;
  - May be dropped;
- Attacker types:
  - Off-Path Attacker: Can't see the victim's traffic;
  - On-Path Attacker: Can see the victim's traffic and add packets with race condition;
  - In-Path Attacker: Can see the victim's traffic, add packets without race conditions and drop packets.

5.3 Local Network Connection

When connecting to a network, a device needs to receive a configuration (IP Addr., Gateway and DNS Addr.) from some other system inside a network.

5.3.1 DHCP (Dynamic Host Control Protocol)

To receive the configuration, our host will broadcast a *Server-Discovery* message that will be answered by the **DHCP** server of the network. The answer will consist of an offer which our host will accept or not and communicate to the server, that will ACKnowledge it. The offer has all the configuration needed.

If someone hears the broadcast and sends an offer before he DHCP server, it could result in big trouble since they can redirect us to any DNS and Gateway of their choosing, meaning our connection has been completely hijacked and they have full control of what we are looking for and at. We now have a complete Man in the Middle. This scenario is called the *Rogue Access point*.

5.3.2 ARP

The Link can't see the IP Addr. and that's why we need a MAC address for communication. the **ARP** protocol is to whom we will turn we need to find another device. We send its IP and:

- If the IP is in the local network → receive its MAC;
- Else → redirects us to the Gateway, to whom we will send our packets in order to get routed towards the destination;

5.3.3 *AirPwn*

When the traffic is non-encrypted and the attacker knows to whom the traffic belongs, it can inject packets, meaning they can race the server to which the client is connected for a response. If the attacker wins, the client receives its malicious packets.

5.3.4 WPA2

The WPA2 execution works as the followin sequence of events (C $\rightarrow$ client, AC $\rightarrow$ Access Point, F $\rightarrow$ Key derivation function):

$$
\begin{aligned}
C, AC &: PSK = F(passwd||SSID) \\
AC &: Anonce = generate() \\
C &: Snonce = generate() \\
AC &\rightarrow_{Anonce} C \\
C &: PTK = F(PSK, Anonce, Snonce, MAC_AP, MAC_C) \\
C &\rightarrow_{Snonce||MIC} AC \\
AC &: PTK = F(PSK, Anonce, Snonce, MAC_AP, MAC_C) \\
AC &\rightarrow_{GTK||MIC} C \\
C &\rightarrow_{ACK} AC
\end{aligned}
$$

Now both devices have the symmetric PTK (Pairwise Transient Key)! Although all the communication between *C* and *AC* can be encrypted from now on, this doesn't prevent someone that has the password from deriving the key if it knows the MAC of the client because the attacker would be able to see all the rest! Even if it doesn't know the PSK, it can still try to brute-force it. This isn't a really safe protocol.

5.3.5 WPA2 Enterprise

WPA2 Enterprise is different. It is much better. The 4-way handshake now happens between the device and an Authentication Server that holds the information required for each user, i.e. each user has an username and a password. This idea makes it so that if someone has access to the internet, it has its own PSK. So, even if the attacker had access to the internet, it wouldn't be able to understand someone else's 4-way handshake. Also, the Authentication Server's connection is encrypted with public-key encryption.

5.3.6 LAN Security

Either we don't provide or we do some *smart switching* and *active monitoring*.

1. **Switch**

   The Switch keeps track of where the MAC addresses inside the local network are seen in order to make things more efficient.

2. **VLANs**

   VLANs are smarter switches. They isolate different parts of the network, making all network traffic inside a VLAN stay inside it.

5.4 Transport Protocols

Before we get into actual communication between IPs and connection resolution for those, we need to clear up what are the transportation protocols for messages and how they work in a pretty abstract way.

5.4.1 UDP (User Datagram Protocol)

Also called "Unreliable" Datagram Protocol, UDP is a lightweight datagram transportation protocol that relies itself in a *best effort* execution, meaning packets can be dropped or arrive in a different and it won't care. Now this clearly seems a really bad way to send a message. The point is: **It is really efficient**.

5.4.2 TCP (Transmission Control Protocol)

The TCP usual communication between two entities are set and terminated goes as the image 1 shows us. Once the communication is set, we can start sending and receiving data. As shown, the connection starts with a Sequence Number from the client as well as another one from the server. The acknowledgment field is always sent as the received Sequence Number plus the amount of data (flag packets are sized one) received and the Sequence Number is always sent as the previous received ACK. The same will happen on data transferring.

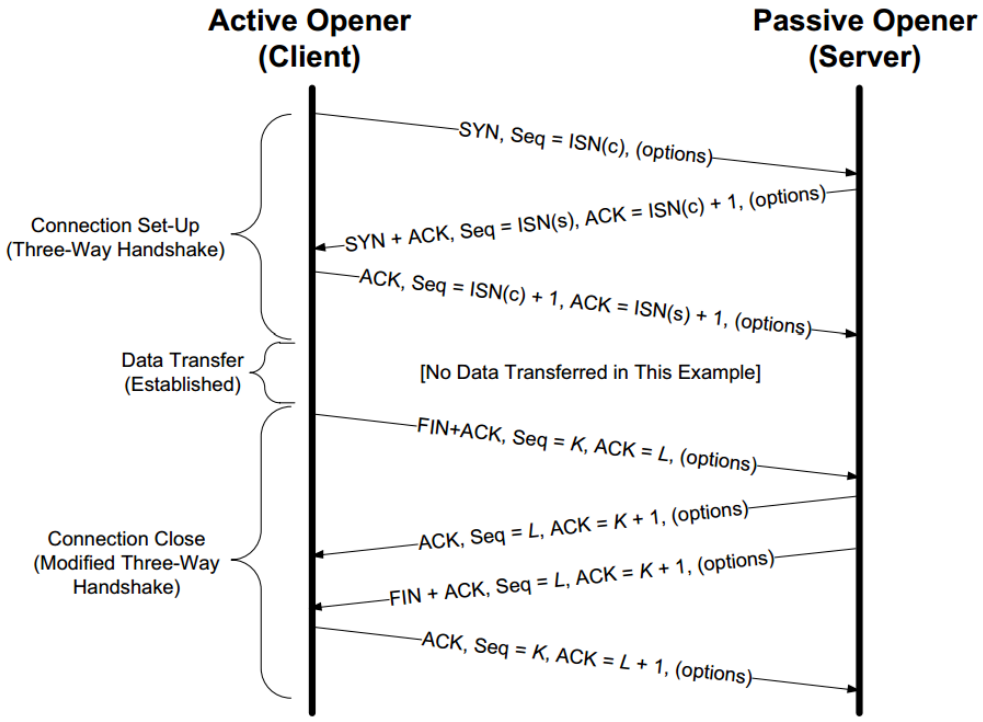


Figure 1: TCP execution

5.4.3 Flags

It's also important to understand what the packet flags mean.

- SYN → Initiate connection;
- ACK → Acknowledge message arrival;
- FIN → Warn that data won't be sent anymore, but the channel is still open for receiving;
- RST → Terminate connection → **Something went wrong!**

5.4.4 Packet & RST Injection

Also, someone who can see the TCP traffic can inject packages. Someone who doesn't can guess the port and Sequence Number. A "RST" packet doesn't need any Sequence Number, so if someone knows the target IP and can guess the port, it can just terminate a connection.

5.4.5 SYN Flooding!

SYN Packets allocate state inside the server. Spamming SYN requests can cause Denial of service. To fix this, we need **SYN Cookies!** The idea behind it is to send a random generated number as the sequence number to the client so that the client has to send it back to you before you actually allocate state. However, we need to assert that the number is the same. For this, we use HMAC. We allocate space, but it's much less than before. The space allocated is for the number we input in the HMAC function. This way we send HMAC(k) and check if the return x provides us with HMAC(x) = HMAC(k).

5.5 DNS

Finds an IP Addr. for communication based on a given URL. Uses UDP because it's performance critical. It's considered a distributed database in which the DNS resolver asks each Name Server for the URL's existance.

5.5.1 DNS Resolver

The DNS Resolver can be Reursive or Iterative. Recursive: Local DNS server → Root (.) → following name server → . . . ; Iterative: Local DNS server looks for Root, that returns the following name server, and then looks for the name server. . .

5.5.2 DNS Response

The DNS Response has the following structure:

Question	Request
Transaction ID	Identification
Answer	If there is any, the requested IP
Authority Section	Name Servers responsible for the answer or the following lookups
Additional Section	Useful info to be cached (possible following lookups)

All sections besides "Transaction ID" are consisted by Resource Records. RRs are structures that contain the following values:

- Hostname → The hostname it represents;
- TTL → Time to live;
- Family → Always going to be "IN" (Internet);
- Type → Type for the value:
  - NS → Name Server;
  - A → IPv4 Addr;
  - AAAA → IPv6 Addr;
- Value → The value it holds

5.5.3 Bailiwick Check

Name server can't add names in the Additional or Authority section for names that aren't in bailiwick, meaning they aren't part of the lookup host. This is to avoid that the Name Server for "mit.edu" adds "berkeley.edu" to the Additional Section with some malicious IP address when someone looks it up. E.g. if you are looking up "eecs.mit.edu" only accept from "*.mit.edu".

5.5.4 Blind Spoofing

This attack involves having someone in a malicious webpage or having them click some bait so that they try to access a specific website, say "bank.com". Once they run the DNS query, the attacker is going to try running against the server response so it may be able to poison the user's DNS cache. To do this, the attacker must be able to spoof the transaction ID, which is random. If it succeeds, the user may have a malicious website instead of "bank.com" and so it goes.

5.5.5 The Kaminsky Attack

Although the idea above seems doable, it's really hard to actually make anything out of it. Consider that you have only one chance of succeeding. Once the value of the lookup is cached, there is nothing we can do about it but wait its TTL. The Kaminsky Attack is a well elaborated alternative to the blind spoofing. In spite of it actually doing mostly the same thing (guessing ID and trying to run), this method, instead of running the lookup to the actual website, looks up fake websites such as "a.bank.com", "b.bank.com",...; Websites that are in Bailiwick and that aren't cached. This way, it still only caches the value when it guesses the correct ID and wins the race, but it can try that as many times as it wants! Some techniques to avoid that is to add CaMeLcAsE letters in the Name Servers and randomize the source port. These two options make spoofing much harder, but still very possible.

5.6 TLS

Built on top of TCP, TLS uses the same ACK/SEQ behaviour. The process is shown by 2 ~~With some english mistakes but I found that on Google Images so I don't care~~. As we can see, the handshake happens by them agreeing on a Cipher spec available for one of the Client's Crypto options. Each one also sends a random 256-bit number (R_C, R_S) to add entropy and avoid replay attacks (isn't shown on image though). The server also sends its certificate for the client to authenticate and start the key sharing process. When the key exchange is done, the data will be communicated in the following format: $\{M, MAC(I, M)\}_C$. Each entity will have its own Cipher and Integrity key to avoid replay attacks with same Sequence Numbers. The key exchange, however, isn't tackled in depth. For that, let's keep going.

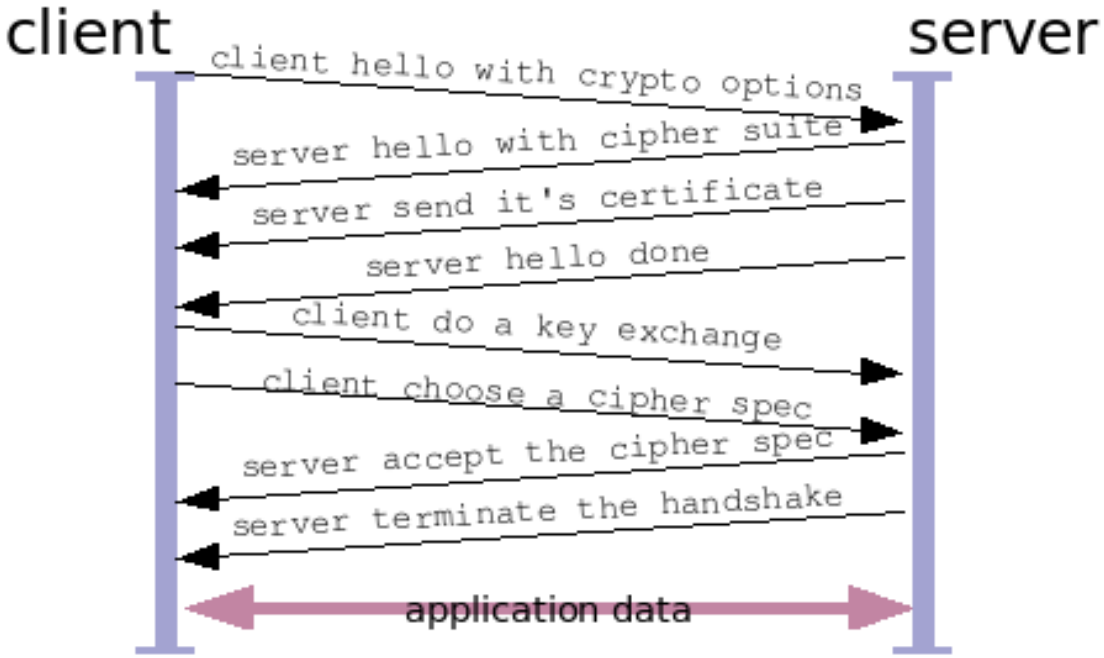


Figure 2: TLS Behaviour

5.6.1 TLS with Diffie-Hellman

Once having its certificate authenticated by the client, the server sends $\{g, p, g^a \mod (p)\}_{K_{server}^{-1}}$. Once the client receives this, the server sends a "end of handshake" message as the client sends $g^b \mod (p)$. Now they both have the *PS* and can derive C_C, C_S (cipher keys) and I_C, I_S (Integrity keys) from {PS, R_C, R_S}. Having all keys, they MAC and send the whole handshake using the Integrity keys to make sure they were derived correctly. Using DH provides forward secrecy, which means that if someone gets ahold of the client or the server's private key in the future, they wouldn't be able to understand the dialog since *a* and *b* were deleted when the conversation stopped.

5.6.2 TLS with RSA Encryption

The difference is that, instead of sending a generated DH public key with its certificate, the server waits for the client to send $\{PS\}_{K_{server}}$ and then all goes along the same way.

5.7 TODO Certificate Authorities - Revised

5.8 TODO DNSSEC

6 Appendix

6.1 Assembly code for Immediate suffering

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

6.1.1 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 [5]:
  - **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

6.1.2 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 [6]:

- 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;

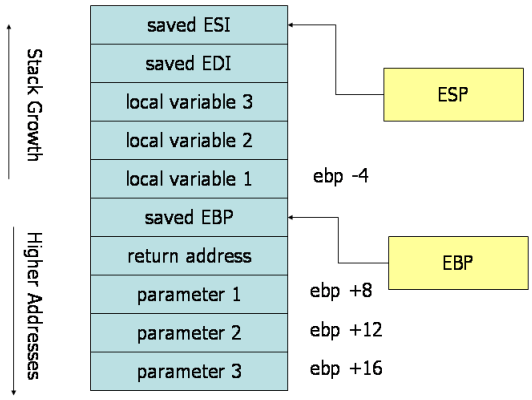


Figure 3: The Stack registers layout

- 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.

6.2 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

6.3 TODO Using GDB & Shell Exploits

6.4 TODO Nick's Insane way of Checking a Lie

References

[1] David Wagner. Security Principles notes, 2019.

[2] Tilo Muller. ASLR Smack & Laugh Reference. page 21.

[3] Nicholas Weaver. Symmetric Encryption slides. Technical report.

[4] Block cipher mode of operation, December 2019. Page Version ID: 928920577.

[5] David Evans. Guide to x86 Assembly.

[6] Zeyuan Hu. Understanding how function call works.