CS 458 — Module 2

Note like, all of the rest of module 2 before is missing since my dumbass wiped my drive by accident.

1 Flaws, Faults, and Failures

- Basically just how flaws are problems, faults are potential problems, failures are occuring problems.
- Unintentional vs intentional security flaws.

2 Unintentional Security Flaws

2.1 Integer Overflows

- The problem is that integers can only represent to a limit (as are other numerical types).
- One common issue is an unsigned integer wrapping into the wrong sign with a huge value!
- Or, casting an unsigned integer into a signed one!
- An attacker can exploit this and pass in values that will trigger overflows that may bypass checks or cause undefined behaviour.
- For example — let's say you pass in a signed int as an argument. The function you pass the value in takes in a signed char. Then you could theoretically overflow it by using a value < -128 or > 127! This could trick some boundary checks, for example:

```
void printchar (signed char i) {
  char *data = "sspppp";
  char size = strlen (data);
  if (i > size - 3) {
        fprintf(stderr, "index␣%d␣not␣allowed\n", i);
        return;
  }
  printf("i␣=␣%d,␣answer:␣%c\n\n", i, data[2+i]);
}

int main (int argc, char **argv) {
  int index;

  if (argc != 2) {
    fprintf(stderr, "Usage:␣%s␣<index>\n",argv[0]); exit(1);
  }
  index = atoi(argv[1]);
  if (index < 0) {
        fprintf(stderr, "index␣%d␣not␣allowed\n", index);
        exit(1);
  }

  printf("Requested␣index␣=␣%d\n", index);
  printchar(index);
}
```

- This code would leak its "secret" key for a value of 255, for example — as it's seen as "-1" by the `printchar` function!

2.2 String Formatting Vulnerabilities

- Unfiltered user input might be used as a formatted string... and this can be really bad!
- For example, using `printf(buffer)` rather than `printf("%s", buffer)`!
- It goes without saying how that first case could be really bad — it may print things that are just lying on the stack!
- Or perhaps something like `printf("%s%s%s%s")` — this might just crash stuff hard.
- `"%x"` will dump parts of the stack... or even better, `"%n"` will *write* to an address on the stack...

2.3 Incomplete Mediation

- Sometimes, we don't trust what users type in.
- For example, maybe they included an extra digit in a form, or a malformed email address using a pound sign rather than an @ sign.
- The application needs to ensure what the user inputs is meaningfull and not incorrect — this is called mediation.
- *Incomplete* mediation is when the application accepts data that is incorrect — for example, bad sanitization can causes cases of XSS!
- In some cases, this is hard to avoid — for example, how do we know if a user's input phone number is a typo or intentional?
- Most of the time, applications will focus on catching entries that are obviously wrong — for example, if someone swaps the date and month fields and passes in the 31st month, then clearly we can inform the user.
- We also need to ensure that any user-supplied inputs are within values that are safe — for example, we shouldn't use a user input if it causes a buffer overflow, or an SQL injection!
- As we mentioned before, XSS (cross site scripting) can be a very common example of this going horribly wrong.
- Many websites have forms that do client-side mediation — then code will validate your data before you submit it.
- But what if a user turns off JS? Edits the form right before submitting it? Connects to the server manually? Modifying the client-side state? Then client-side mediation fails!
- For example, what if a store keeps the unit price for storing a user's order in the client side — and the user changes the value before submitting the form?
- You should *also* do server-side mediation, regardless!
- Assume the client has mucked something up; always check values of all your fields, assert the client has not modified the data in any unallowed way!

2.4 TOCTTOU

- TOCTTOU — time-of-check to time-of-use.
- This is usually known as a race condition error.
- For example, say a user requests the system to perform an action. The system then verifies the user can do this action. Then the system does the action.

- But what if the state changes halfway through the 2nd and 3rd step, and it is not an atomic action?
- What could happen in setuid, which runs with SU privileges, is that an attacker could make a symlink between one file and to a file they owned, get themselves checked, then between the check and open actions, change the link to point to the `/etc/passwd` file. Oops!
- In other words, it didn't check if the user actually opened the file that was checked — it only cared that the file name checked matched the file name opened!
- The main way to prevent this is to check the request and the final action and see there is no mismatch. Locks can also be used to ensure the requested object cannot change. Lastly, acting on the object itself —- the file handle directly — is better than through something that may be indirectly changed (operating on the file names, for example).

3 Malicious Code

- Types of malware:
  - Viruses: malicious code that adds itself to benign programs/files; contains code to spread and attack and are usually activated by users.
  - Worms: malicious code that can spread with little to no user interaction.
  - Trojans: malicious code hidden in software that seems to be innocent.
  - Logic bombs: malicious code that hides until a condition is triggered (ie: time bomb triggers at a specific time).

3.1 Viruses

- A virus infects files that can contain executable code or programs.
- Thus, it spreads when the file is executed/opened.
- The virus "infects" a file by adding code/macros that will execute at the start of the target's program code.
- As mentioned, usually contains code to spread and attack.
- The payload, or what it uses to attack, will usually do some harm to the infected machine... usually something bad.
- So how do we spot viruses?
- When? We should scan when we add files, and from time to time, we should check to ensure we didn't miss anything.
- But how? We can try to check by signature or behaviour.
- This is actually an undecidable problem — there is no ideal solution!
- So assume there is an AV V that is perfect. Then, one could write a program, P^*, which will run $V(P^*)$, then it can just exit. Otherwise, it can just infect.
- In other words, then if $V(P^*)$ is true, then it thought it was a virus, but then it just exits, which means it actually didn't have any malicious behaviour.
- And if it is false, then V thought it wasn't a virus, but then it will execute its malicious code — thus it actually *was* a virus!

- A way of doing this is to just encrypt most of the virus code — the virus code will actually start with a decryption routine to decrypt the rest of the virus.

- Another way of hiding is using instructions that are equivalent to other instructions but different enough, or padding with NOPs. While this would be mostly easy to defeat, the idea of adding enough variations makes scanning slower and harder.

- Behaviour-based systems look for specific patterns of behaviour rather than only a signature. Furthermore, they might run code in a sandbox first.

- Scanners can have two types of errors: false positives or false negatives.

- False positives are potentially even worse — they may lead to users ignoring warnings, or outright disabling the system!

- Base rate fallacy — it may seem like there are more false positives then actual correct detections!

3.2 Worms

- First worm was the Morris worm, which would try to infect systems in three ways:

  1. Exploit a buffer overflow in the "finger" daemon.
  2. Use a backdoor left in the "sendmail" daemon.
  3. Try a dictionary attack against local users' passwords, and if successful, spread to other machines.

- The Code Red worm exploited a buffer overflow in the MS IIS web server for a patch.

- Slammer performed a DoS attack by exploiting a buffer overflow in MS's SQL Server.

- Conficker was a worm that had multiple variants and would use infected machines as part of a botnet (stopped by generating and sinkholing appropriate domains).

- Stuxnet was allegedly created by US and Israeli intelligence agencies, and would target Iranian uranium enrichment centrifuges.

  - A very sophisticated worm; had to be installed through air-gapped systems via USBs.
  - It was a very stealthy and targeted worm; would hide and only unleash its payload if the centrifuges were being used.

3.3 IoT Malware

- With more and more things connecting to the internet, this means there are more and more things that could be targeted or affected by malware (directly or indirectly).

- Furthermore, the devices themselves are often created with security not in mind, and thus easily attacked.

- For example, Mirai used IoT appliances as part of a botnet.

3.4 Trojan Horses

- Trojan horses gain control by getting the user to run code; usually this is code that the user wanted to run.

- Beyond obvious examples (like malicious programs overall), an example is scareware — they may pretend to be a virus scanner and claim that a system is infected!

3.5 Ransomware

- Demands a ransom to restore some hostage resource (usually stuff like decrypting encrypted files).
- Most recent big example was WannaCry, which exploited a Windows SMB vuln that was actually known for a while by the NSA.

3.6 Logic Bombs

- Usually the trigger will be something that the creator can affect in some way — that is.

3.7 Spotting Trojan Horses and Logic Bombs

- Spotting these is really hard, just due to how they work.
- One way of dealing with them is just mitigating the damage they can do.

3.8 Web Bugs

- Usually some object that your browser will fetch (ie: an image that's small). This will mean your browser reveals information about you — IP, contents of cookies, any personal information the site has.
- But why are these malicious? It's mostly an issue of *privacy* — the web bug instructs your browser to work in a way that you might not have expected.
- One could argue this is like a BO attack instructing your browser to behave in a way that you may not expect in terms of security.

3.9 Back Doors

- A set of ways to bypass normal auth. mechanisms that allow access to the system to anyone who knows about it.
- They might be left in for a variety of reasons:
  - Forgot to remove them
  - Left for testing
  - Left for maintenance
  - Intentionally left in for legal reasons
  - Intentionally left in for malicious reasons

3.10 Salami Attacks

- An attack made of many smaller, seemingly inconsequential attacks that accumulate.
- Done so it is harder to detect — maybe they skim a few cents off a transaction, clerks slightly overcharging customers, etc.

3.11 Privilege Escalation

- Most systems have differing levels of privilege for different users.
- This attack allows an unintended user to get higher access.
- This can be done by exploiting a program that has higher access privileges to execute what the attacker wants.
- Another way to do it is tricking the system to believe the attacker is a legitimate higher-privileged user.

3.12 Rootkits

- Contains two parts — one to gain unauthorized root/admin higher-privileges and then tries to hide itself.
- They may hide themselves by modifying commands like `ls` to not report themselves, modify the kernel so that user programs cannot see the files, hide any log messages, etc.
- A famous rootkit was the Sony XCP rootkit that Sony audio CDs came as "copy protection".
- The CD contained an `autorun.exe` file that would automatically execute and install a rootkit.
- It tried to make it such that any process that tried to read an XCP-protected CD would get garbled output.
- Then they put out an uninstaller that left a backdoor.

3.13 Keystroke logging

- Almost all of your information comes from the keyboard.
- Thus, a keyboard logger could easily keep track of data like credentials or private information.
- In some cases, people intentionally install things that do log keys, and it comes down to trust to ensure the information isn't sent elsewhere.

3.14 Interface Illusions

- Things that make it so that what you see is not what is actually occurring on the screen.
- An example is clickjacking in browers.
- The Conficker worm would change the "run program" icon and text to the one that you used to open the folder and view files, baiting users into clicking the wrong entry and running the malware.

3.15 Phishing

- Really, an example of an interface illusion.
- For example, make a fake Paypal website.

3.16 MITM

- Keyloggers, interface illusions, and phishing are examples of MITM.
- You're communicating with something else in the middle of the system you're think you're actually communicating to.
- The user will think nothing is wrong as MITM will intercept the communication and still pass it to the intended party, but now they are privy to the details.

4 Non-malicious Flaws

- One way a non-malicious flaw can be exploited is via covert channels. An attacker creates a capability to transfer information that should not be transferred via that channel.
- For example, hiding sensitive data within less sensitive data (ie: steganography).
- Another is side channels. Eve can watch how Alice's computer behaves while processing sensitive data.
- Two examples are reflections and cache-timing.
- Reflections is literally using a reflecting surface to get information. Like a mirror. Or glasses. Yes, this is exactly as it says it is.
- Cache-timing side channel exploits are things like Spectre and Meltdown — abuse how processors use cache access to let a process learn data about another that should not be shared!
- Some other potential attack vectors:
  - Bandwidth consumption
  - Timing computations
  - Electromagnetic emissions
  - Sound emissions
  - Power consumption
  - Differential power/fault analysis

5 Controls Against Security Flaws

- So. . . given all these threats, what can we even do?
- Security should be considered *in all stages of the software development lifecycle*.
- How can we design programs so they are less likely to have security flaws? Some things include:
  - Modularity:
    - Break problem into smaller modules responsible for smaller subtasks.
    - This means it is easier to test, reuse, maintain, analyze, etc.
    - Have low coupling and high cohesion.
  - Encapsulation
    - Make modules self-contained and share information only as necessary.
    - A developer of a module should not need to know how another one is created.
  - Information hiding
    - Internals of the module should not be visible at all to the other modules.
    - Prevents accidental reliance on behaviours not promised, or hinder malicious actions from the developers themselves.
  - Mutual suspicion
    - Always check the inputs from other modules before processing them.
    - Assume that the inputs are untrusted. Validate!
    - Also a defence against malicious behaviours from other modules, such as corrupted data. This prevents the spread of the compromised module.

    - ∗ But this comes at the cost of performance as now you're checking all inputs.
  - Confinement
    - ∗ We can also confine a module by running it in an environment such that it can only access the resources absolutely needed.
    - ∗ For example, run code that you don't trust in a sandbox/VM.
    - ∗ Obviously you can't run everything in a confined manner; this has a cost in speed and potentially functionality.

- Implementation — what can we do here?
  - Don't use unsafe languages like C, or be careful if you do.
  - Static code analysis — these often look for things like buffer overflows, TOCTTOU, flaws, etc. Note they are not perfect; they cannot verify code is perfect, just spot problems, and may result in false positives sometimes.
  - Hardware-assistance — hardware security in things like the processor are being made; for example ARM has pointer authentication.
  - Formal methods — Usually impossible to do in general but in some cases one can try to do so. There are also programs that one can mark-up their code to get it proven (see: seL4).
  - Genetic diversity — prevent things from all using the same code so malware has a harder time propagating.

- What happens if we do find a flaw? How do we practically reduce them?
  - Firstly, it would be good to be able to track down where the flaw was introduced. For example, version control often tracks every user who commits a change, when, and therefore, one can pinpoint when a change was introduced that corresponded to the problem.
  - Preventing this can potentially be done via things like code reviews — get others to take a look at your changes, as a change in perspective often can spot problems! This is often why open source is deemed more secure — more people are likely to look at it.
  - Two examples of code reviews are guided code reviews (walk through your code to someone else, explain) and Easter egg code reviews, which have intentional flaws so reviewers should expect to find something wrong.
  - Another obvious thing to do? Test! Try to break the program, pretend to be the attacker, see if you can make it do unspecified behaviour (remember, we want things to do only what is defined).
  - Two types of testing — black box testing (cannot see the internals) and white box testing (can see the internals).
  - One type of black box testing is fuzz testing, which will send very, very random data into the object, even data that shouldn't be accepted at all.
  - White-box testing is very useful for regression testing as one can make a very comprehensive set of tests that test many internal parts.
  - One other thing that helps is documentation — explain assumptions, design choices, what works/doesn't work, etc. This helps other developers be privy to your past choices and what's going on currently.
  - Finally, there should be a way to push changes that fix flaws that were missed. That is, the software should be maintainable.
  - Have standards. Specify who should do what, what things must pass before pushing code to master, etc. Audit that your procedures are being done correctly (usually with an external company/person).