Portfolio includes:
Secure Software Development,
Web Security,
Injection Attacks & Taint Analysis,
Secure C,
Home IoT Exercise

by Cherie Mai Caloyloy Hansen

December 17, 2021

Home IoT exercise has been made by *Temporary Group A*:

Mikkel Møller Andersen,
Mark Højer Hansen,
Zohra Amini,
Hasan Khan,
Nikolaos Pavlidis &
Cherie Mai Caloyloy Hansen.

1 Secure Software Development

1.1 Your understanding/definition of security

Within class, we have been presented to multiple definitions of security [9]. These definitions are listed below.

1. "*The ability of a system to satisfy its goals in the presence of an adversary.*"
2. "*Ensure the capability of a system to deliver correct service when under attack.*"
3. "*The ability of a system to ensure CIA.*"
4. "*Precautions to protect assets.*"
5. "*Computer security deals with the prevention and detection of unauthorised actions by users of a computer system.*"

All of the definitions are useful, but not all of them would be included in my definition of security. The 1st definition mentions a system, where a visualisation of this matter can be seen on figure 1 below.

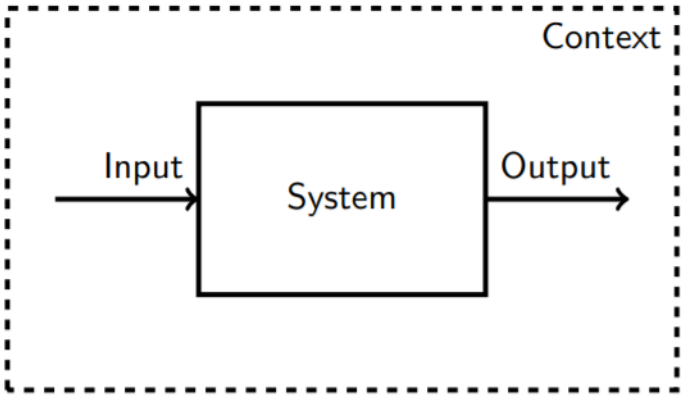

Figure 1: Defining security, based on the class [9]

Figure 1 shows a system with an input and output in a context. What is "fluffy" about this model, is that the system and the context are not defined, the goals and the adversary are not shown in this figure either. Along with the 2nd and 4th definitions, the security is depending on the context. I personally like the 3rd definition more, as the only definition needed is the "system", and CIA have defined goals already. The 5th definition is also good in my opinion, as it further defines *computer* security.

1.2 Your take on risk-analysis/-management

The risk analysis/-management is useful for defining goals and security requirements when implementing software. As the risks are defined before the implementation phase, the software will make use of security-by-design. There are different types of risk analysis, where one from my previous group projects are shown on figure 2 below.

ID	Threat	Why is this a threat?	Estimated consequence	Estimated probability	Calculated risk	Consequence after Mitigation	Probability after Mitigation	New risk
1	DDoS	DDoS attack on the server will result in users being unable to communicate with their TriLock. All use cases will be unavailable during the DDoS attack.	5	2	10	2	2	4
2	Sniffing	In case an attacker is in the vicinity sniffing the TriLock user's network, they could potentially sniff some sensitive data, eg. name, email, phone number, and password.	4	1	4	1	1	1
3	Database hacking	If cyberattackers hack the database, they can steal or leak the personal information of the customers.	5	1	5	3	1	3
4	App hacking	Skilled hackers could potentially reverse engineer the app to find unencrypted data and possibly gain control over database functions.	5	1	5	5	1	5
5	SQL injection	Code injection can give the attacker the ability to manipulate data in the database.	5	3	15	1	1	1
6	Phishing	A phishing attack could be an email from someone pretending to be a friend, family member, or TriLock employee with the purpose of stealing personal data or gaining unauthorized access to their TriLock.	2	2	4	2	2	4
7	Man in the middle attack	An attacker can inject malicious packets into the communication.	3	2	6	3	1	3
8	Database Trojan	If a Trojan is injected into the database, it can perform unauthorized actions to the data such as reading, updating, or deleting data.	5	1	5	5	1	5

Figure 2: Risk Analysis from my P5 report [3].

There are simpler versions of a risk analysis/-management table, where a 3x3 table shows the *difficulty* on one axis and the *impact* on the other axis. The risk analysis on figure 2 is much more detailed and shows both a risk assessment along side risk management. This helped us both find requirements for our system and understand the risks that our system were vulnerable to.

1.3 Your take on a secure software development (SSD) process

To go through an SSD process, I will focus on the Seven Touchpoints. This process is useful for building security in an implementation. The book orders the touchpoints by effectiveness [7], which will be further elaborated in the phase-description below.

1.3.1 The phases of your process

1. Code review: Use analysis tools to find details in the code and eliminate bugs [7][9].
2. Architectural review: Check the implementation so far; how is the system context? Did the security goals cover all the threats [7][9]?
3. Penetration testing: Test and target the security of the code. This phase might find vulnerabilities not found in the Architectural review [7].
4. Risk-based security testing: Similar to penetration testing, the risk-based testing are testing the abuse cases (see below) and security goals [7][9].
5. Abuse case: Describing attack patterns to make requirements for the design of a system [7][9].
6. Security requirements: Making requirements out of security goals. The requirements are also found from the abuse cases (see above) [7][9].
7. Security operations: Going out to the network, understanding how the system operates for other people [7][9].

1.3.2 The "why" of your process

The SSD process is useful when developing software in general, but for my case; my project at Aalborg university. Many methods seen in the *Building Security In* book are similar to ones I have been using before. For instance Use Cases (see next section), Risk analysis (see previous section) and attack trees (mentioned at the comparison section). These methods are/have been useful for many reasons. Why did we use this process? To visualize our risks. To secure our software. To gain requirements that protects the system from attacks.

1.4 How to implement an SSD process into a specific project

As mentioned already, I have had multiple projects where I included the same theories or similar. The seven touchpoints mentions abuse cases and security requirements; similar to my P5 project, we made use cases and set software requirements based on this. Figure 3 is just one use cases out of multiple, but highlights the difference of an authorized user, and someone who is not registered, as they should be able to do different types of actions. The (security) requirements were then defined, as listed:

1. The user must be able to log in.
2. The user must be able to log out.
3. A new user must be able to register an account.
4. The user must be able to register a lock.
5. The user must be able to verify their device as their trusted device.

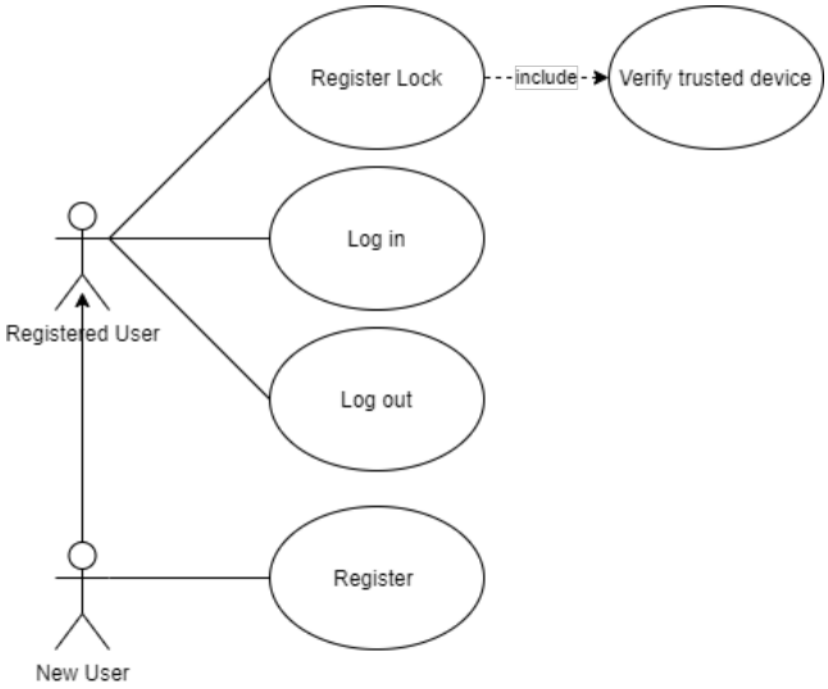

Figure 3: Use Case example from my P5 report [3].

1.5 Comparison with other approaches

In class, we also discussed theories as STRIDE and attack trees. These are not "a process" in the same way, where STRIDE can be seen as a type of security checklist and attack trees can have different goals [9]. In our current project, we make use of attack trees as seen on figure 4, with the goal of understanding/visualizing the risks and attacks of our implementation. Attack trees can also define requirements and used for cost analysis - some trees are made to understand which attacks have the biggest costs [13].

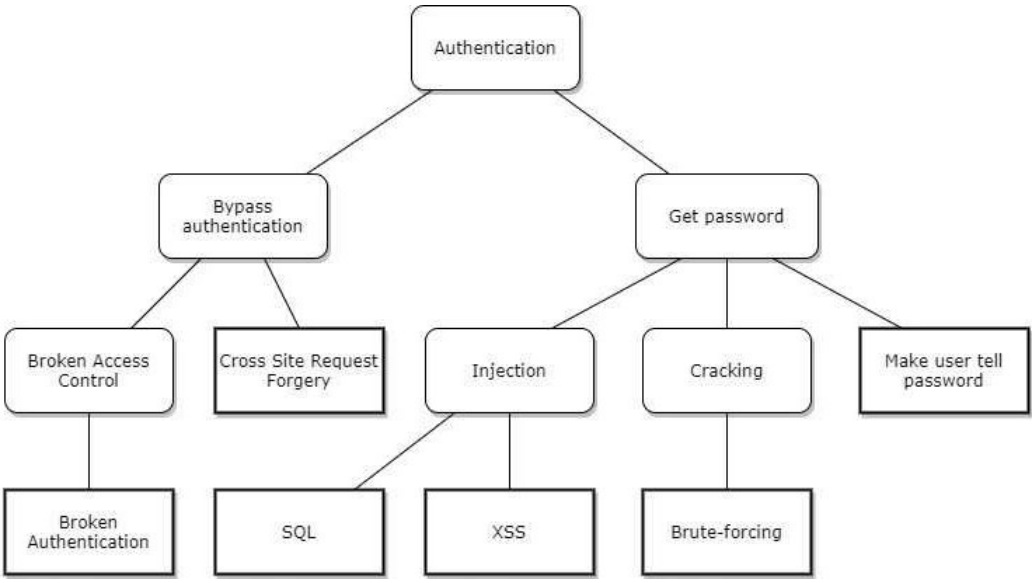

Figure 4: Attack tree which we made for the current project paper.

2 Web Security

2.1 Brief Description if OWASP Top 10 (with examples of attacks)

2.1.1 Broken Access Control

If permission is restricted, for instance because of different levels of privileges, then standard users might want to access data/web pages they initially are not allowed to enter. The principle of least privilege is promoted to make sure users should not access confidential data, which broken access control violates [14]. We do not want attackers and intruders to modify our data.

2.1.2 Cryptographic Failures

Cryptography makes data unreadable, which makes it more secure against adversaries. Failures in cryptography can be caused by old or weak algorithms, or no encryption at all [15]. Weak encryption could be AES-ECB where the encrypted data is visible if multiple blocks are similar as seen on figure 5 below.

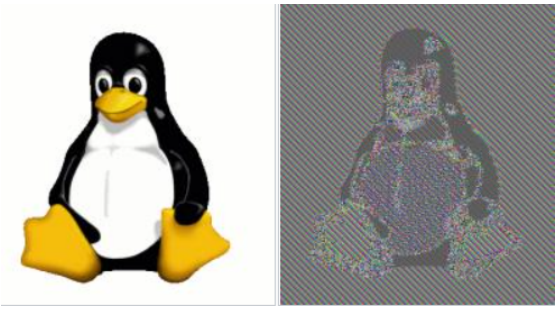

Figure 5: Original Photo & ECB Encrypted Photo [24].

With cryptographic failures being on the 2nd spot within OWASP Top 10 shows the vulnerability of sites without cryptography and importance of web protocols as HTTPS.

2.1.3 Injection

There are different types of injection, including SQL injection [16]. An SQL injection attack is shown below on figure 6, which is an example from class where the attack is performed to ignore further code to be executed after this inject.

`' OR 1=1 --` in the username input resulted in a login, without a username and password combination.

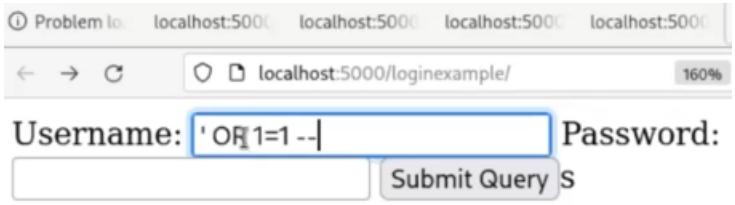

Figure 6: SQL attack shown in class [8].

2.1.4 Insecure Design

When developing software, the developers should have security in mind. If they do not have security in mind, they might not realize vulnerabilities in the system until an attack has been performed. This part is kind of broad and thereby need a lot of resources to secure fully [17]. The *Building Security In* book would make the developers consider the design phrase; make some abuse cases, risk analysis, etc. to ensure the design is better prepared for attacks.

2.1.5 Security Misconfiguration

Misconfiguration of software means error in the implementation or missing code to make it run as intended. Error in code could occur by for example having unnecessary features enables or installed. Missing security could be making the users change a password; if we keep using default accounts and passwords, then it is easier to enter confidential data using automated brute force methods [18].

2.1.6 Vulnerable and Outdated Components

As the title says, the components are not secure if they are vulnerable and outdated. Outdated components are possibly not updated and thereby missing newer and more secure software. If these components are out of date, they might not be supported either, making the system more vulnerable [19].

2.1.7 Identification and Authentication Failures

When you have a login, you wish to be the only person to access this login - this is done with help from authentication. With identification and authentication failures, an adversary might be able to brute force their way into your account, especially with weak or default login and password combinations. To prevent this vulnerability, multi-factor authentication can be used [20].

2.1.8 Software and Data Integrity Failures

When developing software, the code should be secured in terms of integrity - Malicious code should not be able to manipulate the software and tamper the data. The new updates, third-party systems or repositories should not be applied unless it is trusted [21].

2.1.9 Security Logging and Monitoring Failures

Security logging can be used to document the activities in the system, meaning suspicious activity could be discovered if this is applied - But logging also has to be done correctly, as they might be vulnerable if stored locally [22]. Example of log placements could be failed logins, where it can be discovered someone is brute-forcing a login.

2.1.10 Server-Side Request Forgery

Server-Side Request Forgery (SSRF) shows how the adversary can bypass firewalls, VPN, etc. by sending crafted requests to the server. This attack can be done if there is no validation on, for instance, user-supplied URLs - Validation is needed for better security [23].

2.2 Brief description of ASVS

Application Security Verification Standard (ASVS) are used to set a standard for software, and categorize requirements in levels to verify that a system is secure. These verification's levels are set in three different levels - the higher the level, the more secure [12]. Verification Level 1 is just the bare minimum, while Level 2 have more confidential data that needs protection, etc. Level 3 are critical systems that needs the most security.

Based on a discussion in our group, we rated websites and other systems in different levels:

Table 1: ASVS Level discussion in the group

Level	**Examples**
Level 1	"Vasketid.dk". (Single player) games. Sport (bowling) websites.
Level 2	Small-medium enterprises. Streaming platforms (not payment aspect). Public cloud storage. Organizations/companies looking for funding. "dba.dk". Hotels.
Level 3	Power Plants. Health Care systems. Private cloud storage. NemID. Banks. Digital voting. Data centers with classified information. Airports. Critical private inforation on Social Media. Police, firefigthers, etc. Sponsored research centers. Stock exchange.
Off-levels	Social Media; Both have public and private data - possibly multiple levels?

2.3 Pick a few requirements and discuss the reasoning behind them

V2.5 Credential Recovery Requirements

#	Description	L1	L2	L3	CWE	NIST §
2.5.1	Verify that a system generated initial activation or recovery secret is not sent in clear text to the user. (C6)	✓	✓	✓	640	5.1.1.2
2.5.2	Verify password hints or knowledge-based authentication (so-called "secret questions") are not present.	✓	✓	✓	640	5.1.1.2

Figure 7: Description of V2.5.2 [2].

We discussed V2.5.2 which is highlighted on figure 7 above. The discussion is summarized in bullet points:

- Social engineering attacks can be used to bypass knowledge-based authentication (KBA).
- The KBA can also be easily guessed if the adversary knows the target.
- Regarding the point above - Friends could easily target a user.
- Friends has for years been a trusted source for Facebook, where the friends were able to recover your account if the password was forgotten [5] - this may show the friends might enter your account easily if they are trusted.

V2.8 Single or Multi-factor One Time Verifier Requirements

Single-factor One-time Passwords (OTPs) are physical or soft tokens that display a continually changing pseudo-random one-time challenge. These devices make phishing (impersonation) difficult, but not impossible. This type of authenticator is considered "something you have". Multi-factor tokens are similar to single-factor OTPs, but require a valid PIN code, biometric unlocking, USB insertion or NFC pairing or some additional value (such as transaction signing calculators) to be entered to create the final OTP.

#	Description	L1	L2	L3	CWE	NIST §
2.8.5	Verify that if a time-based multi-factor OTP token is re-used during the validity period, it is logged and rejected with secure notifications being sent to the holder of the device.		✓	✓	287	5.1.5.2

Figure 8: Description of V2.8.5 [2].

V2.8.5 as shown on figure 8 above was discussed as well. The discussion is summarized:

- The logging will let the user know that someone is trying to re-use a OTP-token.
- The "one-time" authentication should also be for one-time use, even for the user itself.
- The user should be notified if someone is trying to use their login.
- If the user is notified of rejected time-based authentication, then it is possible that someone at this point knows the username/password of the user.

V1.2 Authentication Architectural Requirements

When designing authentication, it doesn't matter if you have strong hardware enabled multi-factor authentication if an attacker can reset an account by calling a call center and answering commonly known questions. When proofing identity, all authentication pathways must have the same strength.

#	Description	L1	L2	L3	CWE
1.2.1	Verify the use of unique or special low-privilege operating system accounts for all application components, services, and servers. (C3)		✓	✓	250

V2.5 Credential Recovery Requirements

#	Description	L1	L2	L3	CWE	NIST §
2.5.4	Verify shared or default accounts are not present (e.g. "root", "admin", or "sa").	✓	✓	✓	16	5.1.1.2 / A.3

Figure 9: Description of V1.2.1 & V2.5.4 [2].

V1.2.1 & V2.5.4 was merged in a discussion, as these are similar. The discussion is summarized:

- We do not want people to access restricted data
- Even administrators should ask for permission
- We do not want data to run in the root
- Should there even be a root?
- Better applications are aware they should not run as root

3 Injection Attacks & Taint Analysis

3.1 A "reminder" / refresher on taint analysis

In class, we started to discussing taint analysis to help us solve problems like seen on snippet 2 below. The problem in this example, is that the input is controlled by the user, and thereby can be used to possibly catch cookies.

Snippet 1: PHP example from class

```
$choice = $_REQUEST['choice'];
Echo 'your choice was: ' . \$choice;
```

To make the example more secure, we should avoid having no input validation or sanitization. To improve it, we should prevent user-controlled input, and use taint analysis [10].

I will show my notes to describe the terminology, but forgive me for my drawing, writing and how unclear it looks on figure 10 below. Beginning from the overview; a data flow analysis is used to showcase the full taint analysis, we then we then see all possibilities in the program, which in this case is illustrated with a control flow graph (CFG) which is flow-diagram looking. As this data flow analysis has its program defined and a CFG illustrated, data flow equations can be added; these are to explain the data in text format, I personally compare it to how I view set theory in math.

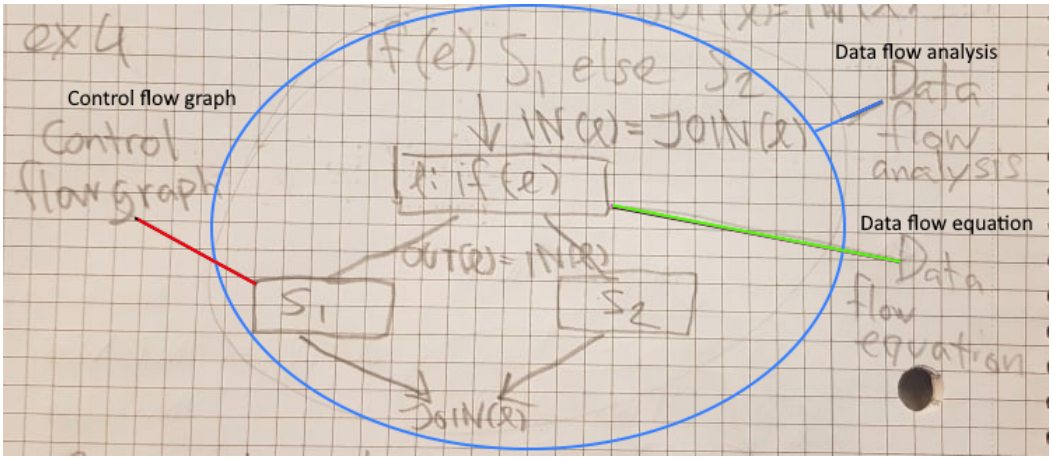

Figure 10: My notes regarding the terminology

3.2 An example taint analysis of a small program

I will use exercise 13 from class, which is shown in figure 11 below. The taint analysis' layout is based on the lecture notes [4].

Exercise 13

Use the static taint analysis on the following program:

```
1  x = input();
2  if(x)
3    y = 1;
4  else
5    y = 42;
6  output(y);
```

Figure 11: Exercise 13.

Figure 12 shows the taint analysis specified for exercise 13, but to see a more general CFG, $\ell : if(x)$ would be replaced with $\ell : if(e)$, $y = 1$ and $y = 42$ would be be replaced with $s1$ and $s2$ as seen on figure 10 in the section above.

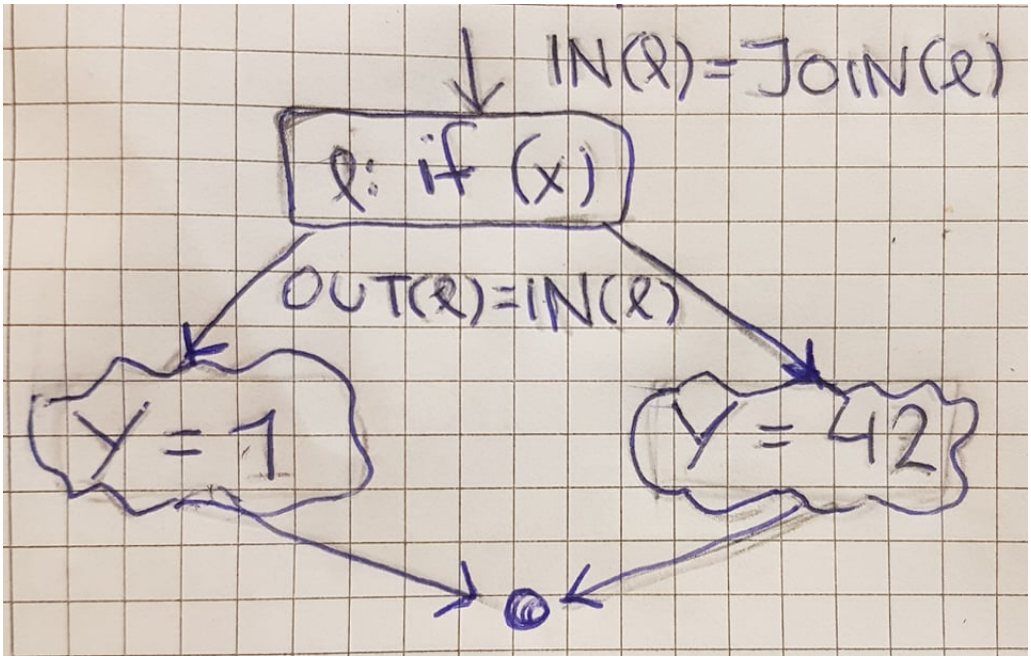

Figure 12: Taint analysis of exercise 13

3.3 Reflections on the use and/or limitations of taint analysis (e.g., wrt. injection attacks etc.)

In class, we went through a couple of PHP examples where it was mentioned injection attacks could be performed, as the input is user controlled. As mentioned earlier, snippet 2

is vulnerable to injection attacks, including cookie stealing. General injection attack is discussed in section 2.1.3, as I do not think it a definition is needed in this part of the portfolio, but rather the Web Security part.

Snippet 2: PHP example from class

```
$choice = $_REQUEST['choice'];
Echo 'your choice was: ' . \$choice;
```

3.4 Notes, insights, and reflections on the generalised taint analysis

My understanding of generalized taint analysis was affected by my knowledge of lattices (see section below). In the lecture notes, the section *Summary: Generalised Taint Analysis* have some notation I am not sure of, but the table of statements can be reflected.

Statement	$IN(\ell)$	$OUT(\ell)$
$\circ$	$\lambda\_.\bot$	$IN(\circ)$
$\bullet$	JOIN($\bullet$)	$IN(\bullet)$
ℓ: `skip`	JOIN(ℓ)	$IN(\ell)$
ℓ: x `=` e	JOIN(ℓ)	$IN(\ell)[x \mapsto taint(IN(\ell), e)]$
ℓ: x `= input()`	JOIN(ℓ)	$IN(\ell)[x \mapsto \lightning]$
ℓ: `output(`e`)`	JOIN(ℓ)	$IN(\ell)$
ℓ: `if(`e`)` S_1 `else` S_2	JOIN(ℓ)	$IN(\ell)$
ℓ: `while(`e`)` S	JOIN(ℓ)	$IN(\ell)$
ℓ: `sanitise(`x`)`	JOIN(ℓ)	$IN(\ell)[x \mapsto \checkmark]$

Figure 13: Found on page 23 of the lecture notes [4]

From my understanding, these statements shown on figure 13 are how to interpret the functionalities of the `While` language and CFGs which is used for this course to showcase generalised taint analysis. These can be compared to how the CFGs are used in this course, where the statement in each can be found on figure 14 - the corresponding $IN(\ell)$ and $OUT(\ell)$ can be found in the table on figure 13.

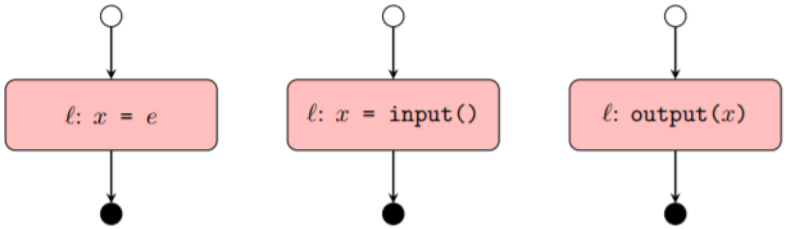

Figure 14: Found on page 10 of the lecture notes [4]

Figure 14 does not show the $IN(\ell)$ and $OUT(\ell)$, but the $IN(\ell)$ would be placed at the white dot (the input) above the statement, and $OUT(\ell)$ would be placed at the black dot (the output) below the statement, and illustrated in figure 15 below.

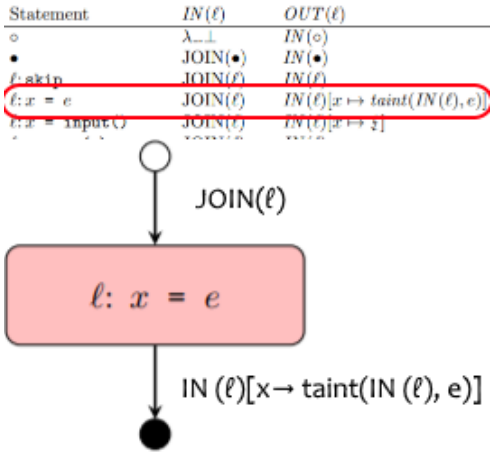

Figure 15: Modified the first statement of figure 14 [4]

3.5 Notes on lattices

Based on my notes, I do not think we talked about lattices so much. I tried going to the lecture notes on figure 16, which pointed me to a direction to understand lattices, but ended up with a one line explanation on figure 17, which I do not think is enough for my understanding. I can admit, I did not make use of lattices in class.

> **Reading recommendation:** If you are not familiar with lattices, consider reading Section 2.11.2 before continuing.

Figure 16: Found on page 19 of the lecture notes [4]

2.11.2 Why lattices?

$\sqcup$	$\bot$	✓	↯
$\bot$	$\bot$	✓	↯
✓	✓	✓	↯
↯	↯	↯	↯

This operator is known as the *least upper bound* operator, or simply "lub" (or "join").

$$\mathsf{TaintAnalysis} = \mathsf{Var} \to \mathsf{Taint}$$

$$\forall f, g \in \mathsf{TaintAnalysis}\text{: } f \sqsubseteq g \quad \text{iff} \quad \forall x : f(x) \sqsubseteq g(x)$$

Figure 17: Lattices part in the lecture notes [4]

3.6 Notes on how to design a (taint-like) analysis

The lecture notes [4] mention that as long as the analysis is designed to possibly tell if data may be tainted, it works.

What we have seen in class has been through visual graphs, mathematical equations and code. Using only one of these methods should be sufficient to make a taint analysis, even if all combined make a better result.

4 Secure C

4.1 How/why C is so difficult to use securely

From what we have seen in class, there are different parts that can make the code (in)secure. During taint analysis class, we went through the problem of user input and how they can be malicious. Different kinds of values can also be overwritten in the stack, which causes problems when executing the code. C is different than other languages, for instance because of its use of pointers, which will be discussed in this section.

4.1.1 The user input is not controlled

An example of code where the user input can be used in a malicious way can be seen in snippet 3 below [11]. The length of `FOO` (see line 4) is not checked, the input does not have a limit in characters, meaning it could be bigger than the size of `buffer` (see line 3). To use it maliciously, could give an input that is bigger than buffer size.

```
1 int main (int argc, char * argv[]) {
2 char outbuf [512];
3 char buffer [512];
4 sprintf(buffer, ``FOO %s'', argv[1]);
5 sprintf(outbuf, buffer);
6 }
```

Snippet 3: no length check on argv

Snippet 4 *should* be more secure, as there is a fixed 400 character limit that will be read and written from `argv` to `buffer`, which is lower than the 512 character sized `buffer` (see line 4).

```
1 int main(int argc, char * argv[]) {
2 char outbuf [512];
3 char buffer [512];
4 sprintf(buffer, ``FOO %.400s'', argv[1]);
5 sprintf(outbuf, buffer);
6 }
```

Snippet 4: now with a length of 400 characters

The input from the user might be limited, but it is still not secure; the user could add characters as "%" which may affect how the code is executed. For instance the input showcased in class, where it manipulates the value to change the input value to a higher limit, breaking the 400 character limit:

%508d\xAA\xAA\xAA\xAA<nops><shellcode>

where `AAAAAAAA` is the shellcode address [11].

4.1.2 Values can be overwritten

When the title says that values can be overwritten, it includes different kinds of values; buffers, integers, canaries (more on canaries in section 4.2.1). The issues regarding buffers is commonly known as buffer overflow, which will be highlighted in the next example snippet 5 [11].

```
1  int main (){
2      long val = 0x41414141;
3      char buf [20];
4      /* ... */
5      scanf(``%24s'', &buf);
6      /* ... */
7      if(val == 0xdeadbeef){
8          setreuid( geteuid(), geteuid());
9          system (``/bin/sh'');
10     }
11     else {
12         printf(``WAY OFF !!!!\n'');
13         exit(1);
14     }
15     return 0;
16 }
```

Snippet 5: narnia0.c

The buffer overflow is an attack in the stack (memory address), where the buffer has a fixed size of 20 characters (see line 3). The problem is when we scan 24 characters (see line 5), where the `buf` will overwrite the `val` in the stack, which will result in filling your memory with too much data [11]. Integer overflow is another example of overwriting values, which will be further discussed in section 4.2.2.

4.1.3 How C is different from other languages

We had a discussion in class, where C was compared to languages like Java and Python - How come these problems are not seen in Java or Python?

Pointers seemed to be the biggest problem we discussed, as they only appear in C. And example of how the pointers makes a difference, is how they access the memory; in Java or Python memory management is in use which will handle the memory automatically - in C, we can go out of bounds of an array possibly reading or writing information we do not want (or crash the program), where it would throw an `ArrayIndexOutOfBoundsException` if it were to execute in Java.

4.2 A few of your favourite C vulnerabilities and ideas for mitigating those

My favorite examples of vulnerabilities we discussed in class are buffer overflow and integer overflow. There are multiple mitigations for these vulnerabilities, but I will only mention what I found interesting from the discussion class.

4.2.1 Buffer overflow

Buffer overflow is already mentioned in section 4.1.2, where a snippet example was shown. A way to mitigate buffer overflow is to use canaries. They are placed in into the stack to protect the return address to be overwritten; it is checked on the return function and will throw an error to indicate data has been overwritten [11]. It was mentioned in the lecture the canary can be seen as "secret/random" information - adversaries would not be able to know if canaries are in use.

4.2.2 Integer Overflow

The snippet is from the JPEG comment field in Netscape, which was shown in class [11]. Snippet 6 below was discussed with a specific example, where we set `len` to 1 (see line 3); if the size is then $len - 2 = -1$, how will it be calculated in binary?

```
1 void getComment (unsigned int len, char * src) {
2     unsigned int size;
3     size = len - 2;
4     char * comment = (char *) malloc(size + 1);
5     memcpy(comment, src, size);
6     return;
7 }
```

Snippet 6: Integer Overflow example

Binary numbers do not show negative numbers; $0001 - 2$ would have to do underflow where it results in 1111 (which is equal to 15 i decimal numbers). Based on `unsigned int` (see line 1), it did look like it was supposed to have non-negative numbers only. Mitigation for this could be to make use of taint analysis.

5 Home IoT Exercise By Temporary Group A

The product we chose is LIXIL Satis Toilet. The Satis Toilet can be seen on figure 18 below.

Figure 18: Satis Toilet [1].

Product ID: identify product name, vendor, vulnerable version(s) of software/firmware/hardware

The vendor is LIXIL Corporation, and the product is Satis version 1.0. The vulnerability was found in 2013, newer versions and products have been developed since [1].

Placement/location/role in home: describe the room(s) in which the product would typically be found; what it is (typically) used for; and the typical user/buyer of the product

The toilet is placed in the bathroom, where it is usually placed towards a wall or into the floor. The technologies of the toilet are controlled with an app, where the user can make it flush, air dry, open/close lid, etc. [1].
The products website advertises towards relaxing homes, comfortable bathrooms, meaning the users are regular people for home use [6].

Define and describe the target system the expected context of the system and what security properties such a system should be expected to have.

The app for the Satis Toilet would be the system. The system will then get inputs from the user who is making use of the app, where the output is the toilet's functionalities which depends on the action/input the user has chosen. The context for using this system is when a user is in need of a toilet (and its functionalities).

Privacy is important as the product will be located in a user's own home. The only device who should be able to connect to the toilet is the user who brought the product, and users who got permission by the product owner.
The system's goal is to provide to the user with a dependable way of controlling the toilet. The security properties focuses on availability. The password is hardcoded in the app, where other devices with the app could manipulate the product. To make sure they can not

access the product, the password should be private and only known for users of the product. The product should make the user change their password, to make sure no intruders can pair with a default factory password. Confidentiality is less important for the product, as the attacker will access user information based on the vulnerability found.

Security/threat model: what aspects of security should the product (have) take(n) into account, what would you expect to be typical attacks/attackers against the product

The security policy should have been that only the product owner (and guests) should be able to control the toilet.

By using the STRIDE model, the typical attacks can be categorized. The typical attacks would include tampering of the toilet functionalities and denial of service attack. An adversary could keep the lid closed, so the user would not be able to use the toilet.

The vulnerability must described in "reasonable" detail, i.e., you do not need to describe a detailed exploit (unless it's short and fun), including how it breaks security (what does security mean here)

The product consisted of the toilet and an app to control the toilet - the app is using Bluetooth pairing and the hardcoded PIN "0000" to connect with the toilet. The attack on the vulnerability was accomplished using the code below, where the PIN code is used to start pairing the Bluetooth device. That actually means that every individual malicious or not can access the toilet by only installing the app to their phones [1].

```
BluetoothDevice localBluetoothDevice =
BluetoothManager.getInstance().execPairing(paramString, ‘‘0000’’)
```

In this context the vulnerability compromises security in the sense that, if an adversary exploits the aforementioned weakness, they can break the availability of the system and thus disrupt the functionality of the service provided by the toilet.

Impact of a successful attack should be described in detail, i.e., What are the consequences (and to whom) of a successful attack: is someones privacy compromised, does it facilitate break-ins or monitoring etc.

The impact of a successful attack on the smart toilet would result in a denial of service, so the system would be available for the user. Furthermore, an attacker could keep flushing the toilet continuously, which would eventually lead to breach of and the build up of a large water utility bill for the toilet's owner.

Bibliography

[1] Trustwave Advisories. *TWSL2013-020: Hard-Coded Bluetooth PIN Vulnerability in LIXIL Satis Toilet*. 2013. URL: https://seclists.org/fulldisclosure/2013/Aug/18.

[2] The OWASP Foundation. "Application Security Verification Standard 4.0.2". In: *OWASP* (2020).

[3] Cherie Mai Caloyloy Hansen, Jonas Bukrinski Andersen, and Lukas Steffensen. "TriLock". Aalborg University of Copenhagen, 2020.

[4] René Rydhof Hansen. "Language-Based Security: A Problem Based Introduction". Aalborg University of Copenhagen, 2020.

[5] Craig Johnson. *Here's how your friends could help you recover your Facebook account*. 2018. URL: https://www.wftv.com/consumer/clark-howard/heres-how-your-friends-could-help-you-recover-your-facebook-account/700762337/.

[6] LIXIL. *Satis*. 2021. URL: https://www.lixil.co.jp/lineup/toiletroom/satis/.

[7] Gary McGraw. *Software Security: Building Security In*. 2006. DOI: 10.1109/ISSRE.2006.43.

[8] Danny B. Poulsen. *WebSecurity 04 - E21*. Aalborg University, Copenhagen. 2021. URL: https://panopto.aau.dk/Panopto/Pages/Viewer.aspx?id=2d62808b-9656-49e2-802a-adb800bef566.

[9] Danny B. Poulsen and René Rydhof Hansen. "Secure Software Development: Course introduction + "Building Security In"". In: Aalborg University, 2021.

[10] Danny Bøgsted Poulsen and René Rydhof Hansen. "Injection Attacks and Taint Analysis: Advanced Software Security". In: Aalborg University, 2021.

[11] Danny Bøgsted Poulsen and René Rydhof Hansen. "ISecure Software Development: (In)Secure C". In: Aalborg University, 2021.

[12] Danny Bøgsted Poulsen and René Rydhof Hansen. "Web Security". In: Aalborg University, 2021.

[13] B. Schneier. *Attack Trees*. 2021. URL: https://www.schneier.com/academic/archives/1999/12/attack_trees.html.

[14] OWASP Top 10 team. *A01:2021 – Broken Access Control*. 2021. URL: https://owasp.org/Top10/A01_2021-Broken_Access_Control/.

[15] OWASP Top 10 team. *A02:2021 – Cryptographic Failures*. 2021. URL: https://owasp.org/Top10/A02_2021-Cryptographic_Failures/.

[16] OWASP Top 10 team. *A03:2021 – Injection*. 2021. URL: https://owasp.org/Top10/A03_2021-Injection/.

[17] OWASP Top 10 team. *A04:2021 – Insecure Design*. 2021. URL: https://owasp.org/Top10/A04_2021-Insecure_Design/.

[18] OWASP Top 10 team. *A05:2021 – Security Misconfiguration*. 2021. URL: https://owasp.org/Top10/A05_2021-Security_Misconfiguration/.

[19] OWASP Top 10 team. *A06:2021 – Vulnerable and Outdated Components*. 2021. URL: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/.

[20] OWASP Top 10 team. *A07:2021 – Identification and Authentication Failures*. 2021. URL: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/.

[21] OWASP Top 10 team. *A08:2021 – Software and Data Integrity Failures*. 2021. URL: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/.

[22] OWASP Top 10 team. *A09:2021 – Security Logging and Monitoring Failures*. 2021. URL: https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/.

[23] OWASP Top 10 team. *A10:2021 – Server-Side Request Forgery (SSRF)*. 2021. URL: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%5C%28SSRF%5C%29/.

[24] Wikipedia. *Block cipher mode of operation*. 2021. URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.