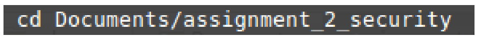
Assignment Two

Gearoid Sheehan

Security for Software Systems COMP8050

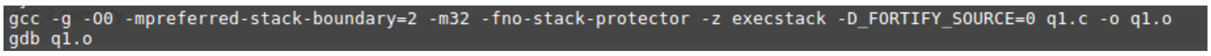
Question 1

Changed directory to where q1.c was saved



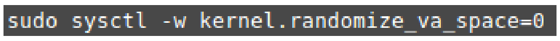
```
cd Documents/assignment_2_security
```

Ran the GCC compiler with the necessary flags etc. Created binary file q1.o. Opened in GDB debugger.



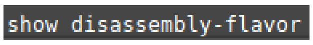
```
gcc -g -O0 -mpreferred-stack-boundary=2 -m32 -fno-stack-protector -z execstack -D_FORTIFY_SOURCE=0 q1.c -o q1.o
gdb q1.o
```

Disabled ASLR.



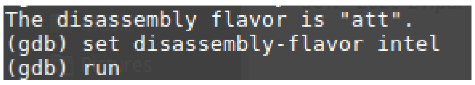
```
sudo sysctl -w kernel.randomize_va_space=0
```

Checked disassembly flavour.



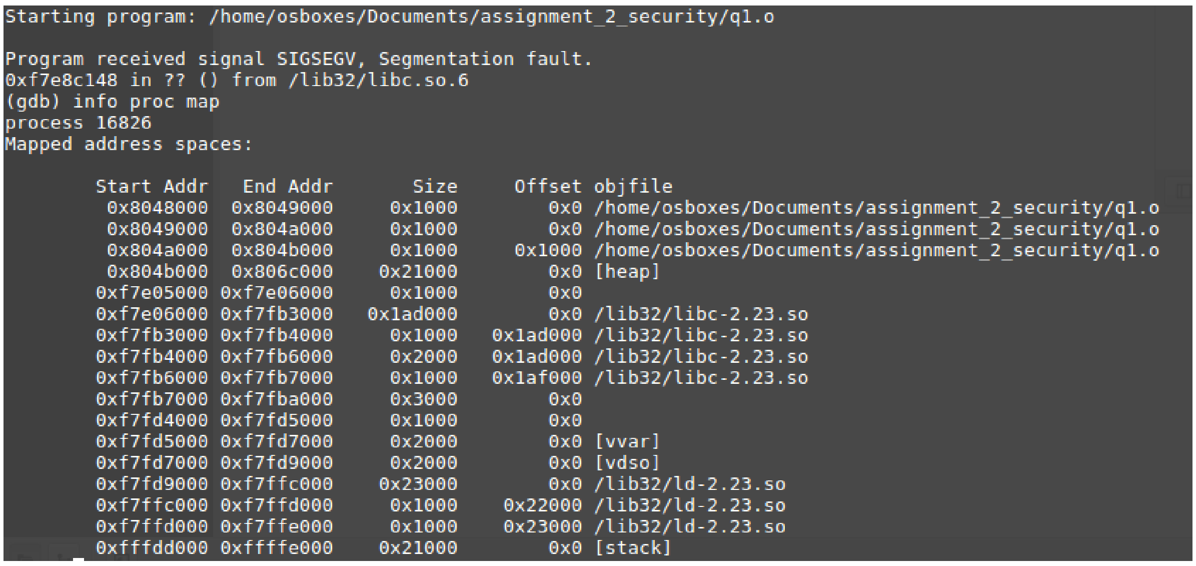
```
show disassembly-flavor
```

Changed from att to intel. Ran the program inside GDB.



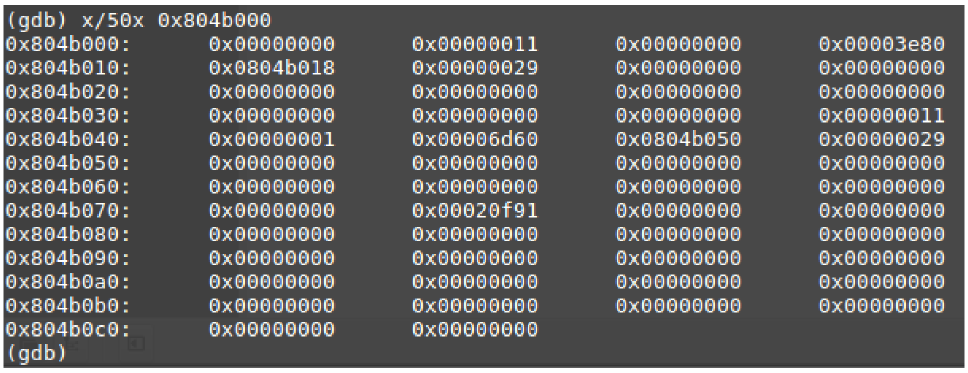
```
The disassembly flavor is "att".
(gdb) set disassembly-flavor intel
(gdb) run
```

As it was run without parameters, a segmentation fault occurred as expected. Here I ran info proc map to show the processes information. I noted that the heap started at address 0x804b000.



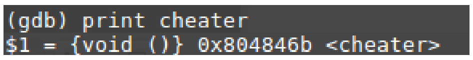
```
Starting program: /home/osboxes/Documents/assignment_2_security/q1.o

Program received signal SIGSEGV, Segmentation fault.
0xf7e8c148 in ?? () from /lib32/libc.so.6
(gdb) info proc map
process 16826
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/osboxes/Documents/assignment_2_security/q1.o
         0x8049000  0x804a000     0x1000        0x0 /home/osboxes/Documents/assignment_2_security/q1.o
         0x804a000  0x804b000     0x1000     0x1000 /home/osboxes/Documents/assignment_2_security/q1.o
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xf7e05000 0xf7e06000     0x1000        0x0
        0xf7e06000 0xf7fb3000   0x1ad000        0x0 /lib32/libc-2.23.so
        0xf7fb3000 0xf7fb4000     0x1000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb4000 0xf7fb6000     0x2000   0x1ad000 /lib32/libc-2.23.so
        0xf7fb6000 0xf7fb7000     0x1000   0x1af000 /lib32/libc-2.23.so
        0xf7fb7000 0xf7fba000     0x3000        0x0
        0xf7fd4000 0xf7fd5000     0x1000        0x0
        0xf7fd5000 0xf7fd7000     0x2000        0x0 [vvar]
        0xf7fd7000 0xf7fd9000     0x2000        0x0 [vdso]
        0xf7fd9000 0xf7ffc000    0x23000        0x0 /lib32/ld-2.23.so
        0xf7ffc000 0xf7ffd000     0x1000    0x22000 /lib32/ld-2.23.so
        0xf7ffd000 0xf7ffe000     0x1000    0x23000 /lib32/ld-2.23.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

Using this address, I printed out the next 50 words on the heap.



```
(gdb) x/50x 0x804b000
0x804b000:      0x00000000      0x00000011      0x00000000      0x00003e80
0x804b010:      0x0804b018      0x00000029      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b030:      0x00000000      0x00000000      0x00000000      0x00000011
0x804b040:      0x00000001      0x00006d60      0x0804b050      0x00000029
0x804b050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b070:      0x00000000      0x00020f91      0x00000000      0x00000000
0x804b080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0c0:      0x00000000      0x00000000
(gdb)
```

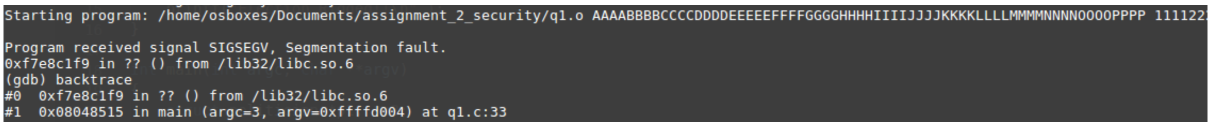
I then found the memory address which the function cheater() began at.



```
(gdb) print cheater
$1 = {void ()} 0x804846b <cheater>
```

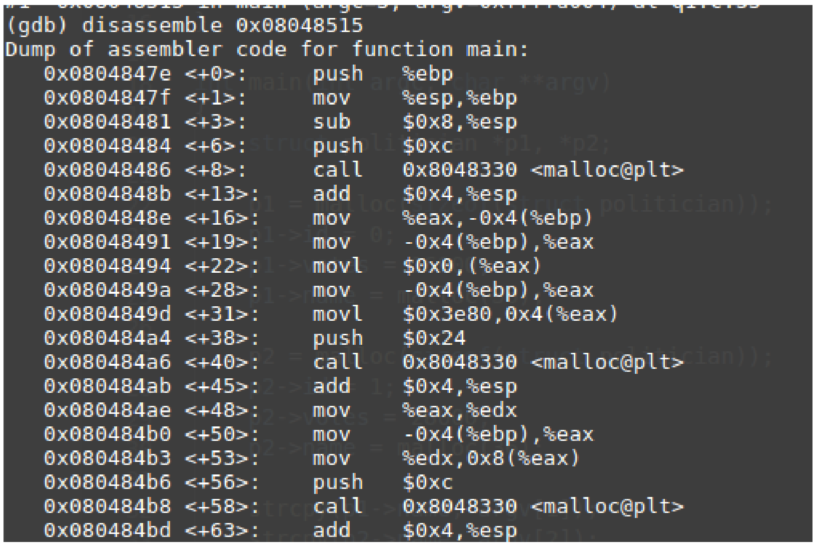
Ran the program with the intention of overflowing the strcpy() functions.

Used the backtrace command to find the location from where we were calling the string copy.



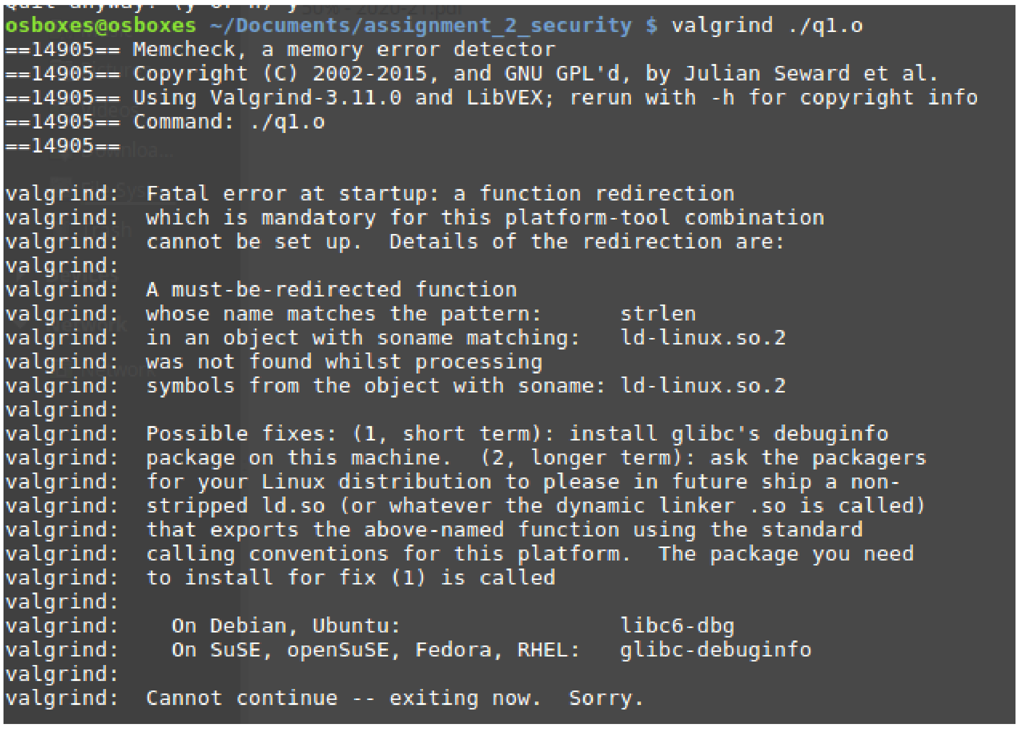
```
Starting program: /home/osboxes/Documents/assignment_2_security/q1.o AAAABBBBCCCCDDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP 111122

Program received signal SIGSEGV, Segmentation fault.
0xf7e8c1f9 in ?? () from /lib32/libc.so.6
(gdb) backtrace
#0  0xf7e8c1f9 in ?? () from /lib32/libc.so.6
#1  0x08048515 in main (argc=3, argv=0xffffd004) at q1.c:33
```

I then disassembled the given address.



```
(gdb) disassemble 0x08048515
Dump of assembler code for function main:
   0x0804847e <+0>:     push   %ebp
   0x0804847f <+1>:     mov    %esp,%ebp
   0x08048481 <+3>:     sub    $0x8,%esp
   0x08048484 <+6>:     push   $0xc
   0x08048486 <+8>:     call   0x8048330 <malloc@plt>
   0x0804848b <+13>:    add    $0x4,%esp
   0x0804848e <+16>:    mov    %eax,-0x4(%ebp)
   0x08048491 <+19>:    mov    -0x4(%ebp),%eax
   0x08048494 <+22>:    movl   $0x0,(%eax)
   0x0804849a <+28>:    mov    -0x4(%ebp),%eax
   0x0804849d <+31>:    movl   $0x3e80,0x4(%eax)
   0x080484a4 <+38>:    push   $0x24
   0x080484a6 <+40>:    call   0x8048330 <malloc@plt>
   0x080484ab <+45>:    add    $0x4,%esp
   0x080484ae <+48>:    mov    %eax,%edx
   0x080484b0 <+50>:    mov    -0x4(%ebp),%eax
   0x080484b3 <+53>:    mov    %edx,0x8(%eax)
   0x080484b6 <+56>:    push   $0xc
   0x080484b8 <+58>:    call   0x8048330 <malloc@plt>
   0x080484bd <+63>:    add    $0x4,%esp
```

Unfortunately, I was unable to make much more headway with this attack as I completed this question last and could not figure it out in time for the deadline. From searching online, I attempted to install a software called Valgrind to my Linux Mint OS, which would have displayed to me the location of where the segmentation fault was occurring. I would have then calculated the exact number of characters needed to over the strpcpy() function. A python file would have been used similarly to the last assignment, and the second parameter would have to be overflowed also with a correct return address to carry out the attack cleanly if I am correct.



```
osboxes@osboxes ~/Documents/assignment_2_security $ valgrind ./q1.o
==14905== Memcheck, a memory error detector
==14905== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==14905== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==14905== Command: ./q1.o
==14905==

valgrind:  Fatal error at startup: a function redirection
valgrind:  which is mandatory for this platform-tool combination
valgrind:  cannot be set up.  Details of the redirection are:
valgrind:
valgrind:  A must-be-redirected function
valgrind:  whose name matches the pattern:      strlen
valgrind:  in an object with soname matching:   ld-linux.so.2
valgrind:  was not found whilst processing
valgrind:  symbols from the object with soname: ld-linux.so.2
valgrind:
valgrind:  Possible fixes: (1, short term): install glibc's debuginfo
valgrind:  package on this machine.  (2, longer term): ask the packagers
valgrind:  for your Linux distribution to please in future ship a non-
valgrind:  stripped ld.so (or whatever the dynamic linker .so is called)
valgrind:  that exports the above-named function using the standard
valgrind:  calling conventions for this platform.  The package you need
valgrind:  to install for fix (1) is called
valgrind:
valgrind:    On Debian, Ubuntu:                 libc6-dbg
valgrind:    On SuSE, openSuSE, Fedora, RHEL:   glibc-debuginfo
valgrind:
valgrind:  Cannot continue -- exiting now.  Sorry.
```

Question 2

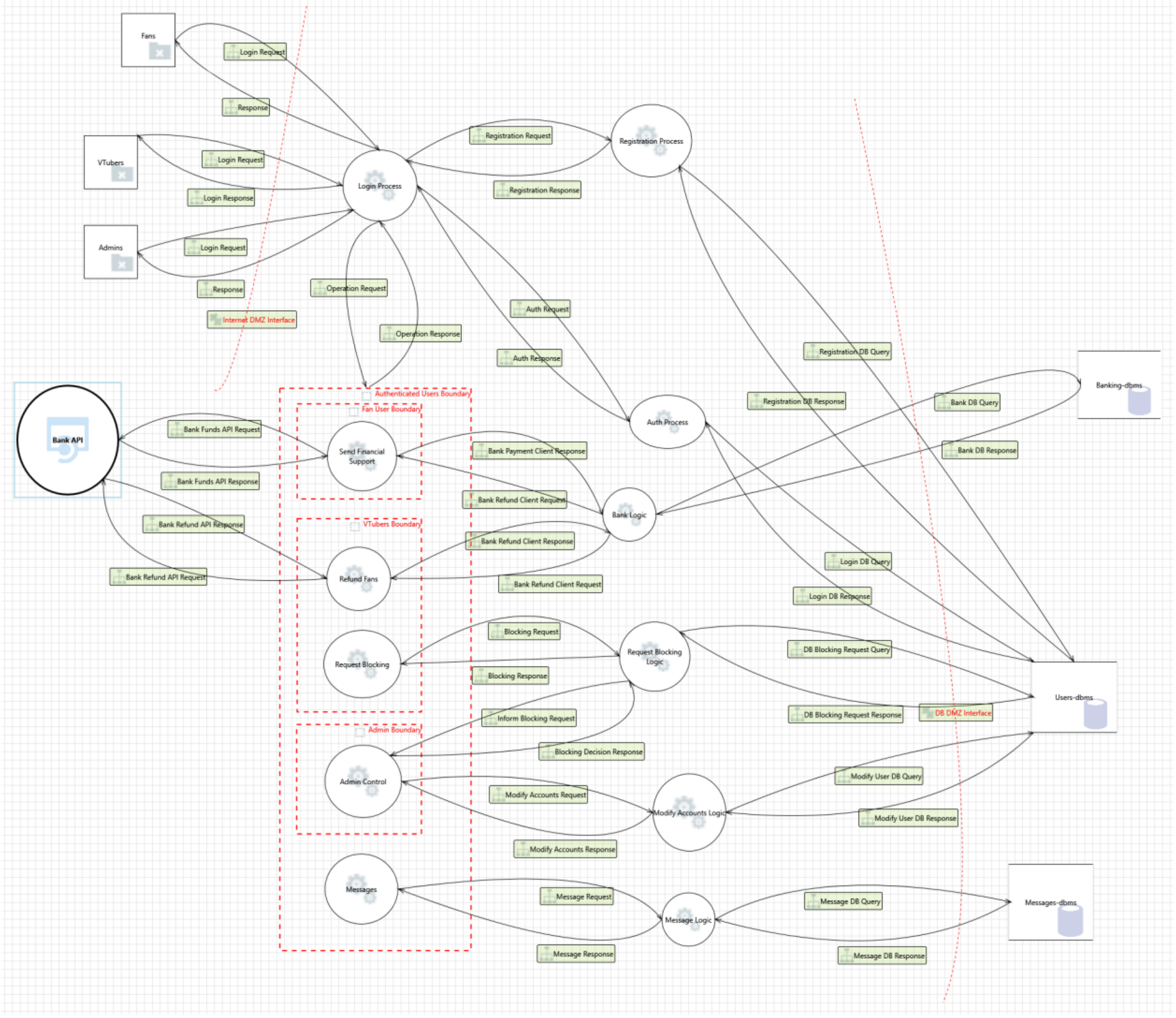
Data Flow Diagram

Threat Modeling Report

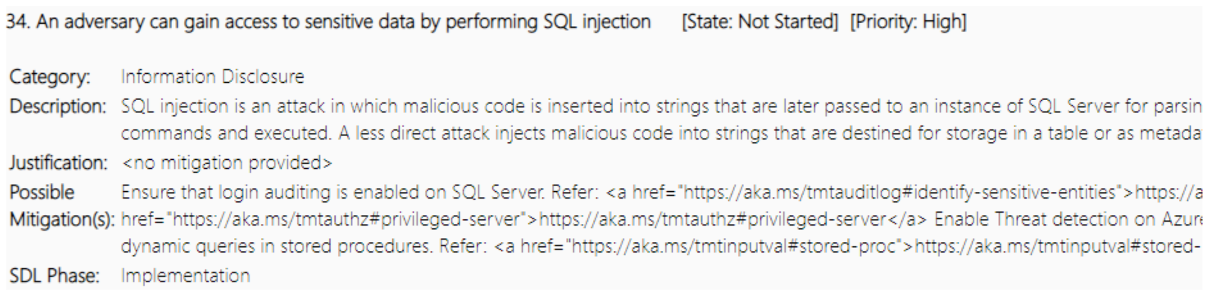
Created on 25/12/2020 21:33:16
Threat Model Name: Hitogata Web Application Threat Model
Owner: Gearóid Sheehan (R00151523)
Reviewer:
Contributors: Gearóid Sheehan
Description: A marketing company has developed a social platform to allow fans to interact directly with Virtual Youtubers through a smartphone app, "Hitogata". The "Hitogata" app serves as a portal to a centralised online system located in the marketing company's HQ building. It allows fans to send messages to their favourite Virtual Youtubers, all of which are permanently recorded in a database, and allows them to send financial support. All such payments are also recorded. Fans may also interact with other fans, and a system is provided to allow fans to organise trades of vtuber merchandise. There are 3 types of uses of the system: fans, vtubers and admins. The app allows customers to send messages to vtubers and receive responses, as well as to view their message history. Similarly payments can be made and reviewed. Credit card details, legal name and proof of identification (e.g. passport id), and bank account details (to receive payments for merchandise trades) are required to register a fan account. Vtubers can view and respond to messages from fans, provide refunds for payments received from fans, as well as monitor any trades relating to their merchandise. They can also apply to the admins to ban any fan account which is behaving too stalker-ish. Admins have complete control over other accounts, being able to create, delete or modify any of their details. All interaction is via the "Hitogata" smartphone app front end. There are at least 2 databases: users-dbms and banking-dbms Users-dbms contains all fan and vtubers' data as well as their message history. Banking-dbms contains credit card and bank information, and transaction histories.
Assumptions: VTubers cannot send financial support to themselves/others. The bank API being accessed has its own security and only needs a fans IBAN and BIC to be passed to it to be used. VTubers can submit blocking requests but must wait until an admin approves the block. Admins can do everything a fan and vtuber can do as well as the admin controls.
External Dependencies:

Threat Model Summary:

Not Started	58
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	0
Total	58
Total Migrated	0



Threat 1



34. An adversary can gain access to sensitive data by performing SQL injection [State: Not Started] [Priority: High]

Category:	Information Disclosure
Description:	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsin commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metada
Justification:	<no mitigation provided>
Possible Mitigation(s):	Ensure that login auditing is enabled on SQL Server. Refer: <a href="https://aka.ms/tmtauditlog#identify-sensitive-entities">https://a href="https://aka.ms/tmtauthz#privileged-server">https://aka.ms/tmtauthz#privileged-server</a> Enable Threat detection on Azur dynamic queries in stored procedures. Refer: <a href="https://aka.ms/tmtinputval#stored-proc">https://aka.ms/tmtinputval#stored-
SDL Phase:	Implementation

DREAD Analysis

Damage Potential: 9

Reason: SQL Injection attacks can result in accounts being breached, passwords being stolen, and sensitive data being deleted or changed. For obvious reasons the damage potential for an application is massive if such an attack was to occur.

Reproducibility: 8

Reason: If the developers wrote code and failed to follow the correct sanitation of inputs and the like where SQL injection attacks can occur, once a vulnerability is spotted by an attacker it can be used over and over again until it is flagged by the applications development team and fixed.

Exploitability: 5

Reason: While it is a very serious vulnerability, it would require an attacker with a certain pedigree of hacking skills and would be difficult for a rookie to carry out.

Affected Users: 7

Reason: The effected users would entail nearly every user of the application. As both admins and other user's login using the same login function, if the attacker logged in as an admin, they would have control of nearly everything on the application.

Discoverability: 6

Reason: SQL injection attacks are a very common type of attack, and those carrying out the attack would attempt to hack the login page as one of their first attempts. However, naturally not all that notice the vulnerability would be able to make use of it

Total Risk Rating: 7

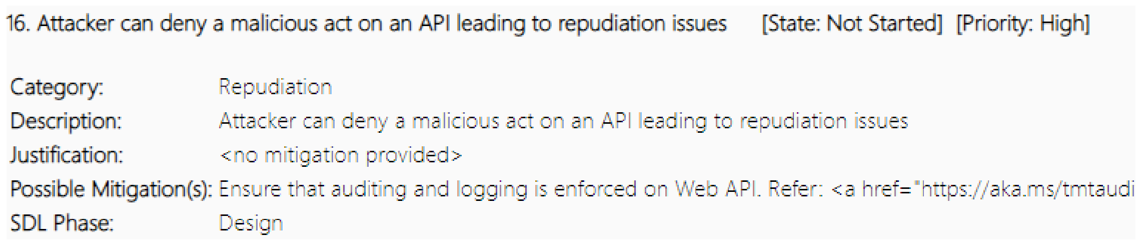
Solution for Developers

Use strong authorization.

Use strong encryption.

Secure communication links with protocols that provide message confidentiality. Do not store secrets (for example, passwords) in plaintext.

Threat 2


16. Attacker can deny a malicious act on an API leading to repudiation issues [State: Not Started] [Priority: High]

Category:	Repudiation
Description:	Attacker can deny a malicious act on an API leading to repudiation issues
Justification:	<no mitigation provided>
Possible Mitigation(s):	Ensure that auditing and logging is enforced on Web API. Refer: <a href="https://aka.ms/tmtaudi
SDL Phase:	Design

DREAD Analysis

Damage Potential: 7

Reason: This occurs in the application when it does not adopt controls to correctly log and track the user's actions. It can be used to change the authoring information of actions executed by a malicious user in order to log wrong data to log files.

Reproducibility: 4

Reason: It can only be attacked when the user is using a feature which calls an external API such as the Bank API

Exploitability: 8

Reason: It requires an extensive knowledge of hacking and would not be able to be completed by an average user.

Affected Users: 2

Reason: The scope of users effected would be only the user accessing the API.

Discoverability: 4

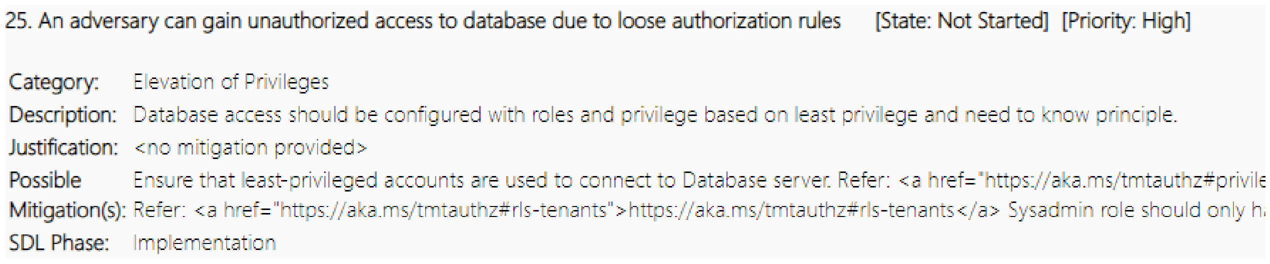
Reason: It is not an obvious choice for rookie attackers to attempt to exploit.

Total Risk Rating: 5

Solution for Developers
Create secure audit trails.

Use digital signatures.

Threat 3


25. An adversary can gain unauthorized access to database due to loose authorization rules [State: Not Started] [Priority: High]

Category:	Elevation of Privileges
Description:	Database access should be configured with roles and privilege based on least privilege and need to know principle.
Justification:	<no mitigation provided>
Possible Mitigation(s):	Ensure that least-privileged accounts are used to connect to Database server. Refer: <a href="https://aka.ms/tmtauthz#privile Refer: <a href="https://aka.ms/tmtauthz#rls-tenants">https://aka.ms/tmtauthz#rls-tenants</a> Sysadmin role should only h
SDL Phase:	Implementation

DREAD Analysis

Damage Potential: 7

Reason: If users can access features which are not meant to be available to their specific role, then tasks such as blocking users which must be done by admins can be used by regular fans which would cause massive problems.

Reproducibility: 9

Reason: It would vulnerable all the time unless correct authorization rules were implemented

Exploitability: 10

Reason: Users would not have to do anything to take advantage of the vulnerability, they would just have to log on as normal and could access admin features.

Affected Users: 8

Reason: This vulnerability would affect all users, and all of them equally as they all would have the same access to everything

Discoverability: 9

Reason: The fan and vtubers would probably realise very quickly that they have access to admin features.

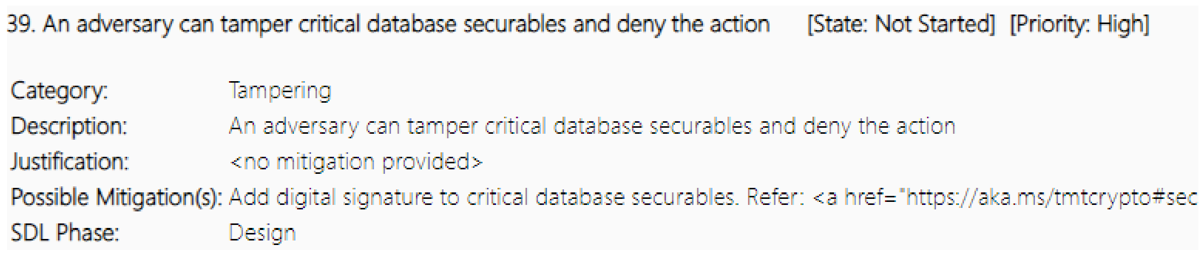
Total Risk Rating: 8

Solution for Developers

Follow the principle of least privilege.

Use least privileged service accounts to run processes and access resources.

Threat 4


39. An adversary can tamper critical database securables and deny the action [State: Not Started] [Priority: High]

Category:	Tampering
Description:	An adversary can tamper critical database securables and deny the action
Justification:	<no mitigation provided>
Possible Mitigation(s):	Add digital signature to critical database securables. Refer: <a href="https://aka.ms/tmtcrypto#sec
SDL Phase:	Design

DREAD Analysis

Damage Potential: 9

Reason: If the database entries are being tampered with, the whole application is at risk of being compromised.

Reproducibility: 7

Reason: It would vulnerable all the time unless correct digital signatures were implemented

Exploitability: 3

Reason: It would require experienced hackers to exploit the vulnerability.

Affected Users: 8

Reason: This vulnerability would affect all users.

Discoverability: 2

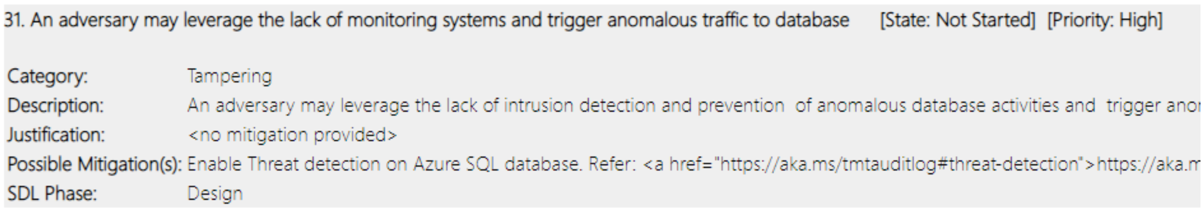
Reason: It would require hackers who were specifically looking for this vulnerability.

Total Risk Rating: 5

Solution for Developers

Use data hashing and signing.
Use digital signatures.
Use strong authorization.
Use tamper-resistant protocols across communication links.
Secure communication links with protocols that provide message integrity.

Threat 5



31. An adversary may leverage the lack of monitoring systems and trigger anomalous traffic to database [State: Not Started] [Priority: High]

Category:	Tampering
Description:	An adversary may leverage the lack of intrusion detection and prevention of anomalous database activities and trigger anor
Justification:	<no mitigation provided>
Possible Mitigation(s):	Enable Threat detection on Azure SQL database. Refer: <a href="https://aka.ms/tmtauditlog#threat-detection">https://aka.m
SDL Phase:	Design

DREAD Analysis

Damage Potential: 6

Reason: Anomalous traffic to the database would end up in incorrect data populating tables, for example login details.

Reproducibility: 8

Reason: It would vulnerable all the time unless correct authorization rules were implemented

Exploitability: 3

Reason: It would require experienced hackers to exploit the vulnerability.

Affected Users: 8

Reason: This vulnerability would affect all users.

Discoverability: 2

Reason: It would require hackers who were specifically looking for this vulnerability.

Total Risk Rating: 5

Solution for Developers

Use data hashing and signing.
Use digital signatures.
Use strong authorization.
Use tamper-resistant protocols across communication links.
Secure communication links with protocols that provide message integrity.

Question 3

1) Cross-site scripting, or XSS for short, is a code injection attack which is executed on the client-side of a web-application. In simple terms, it is when an attacker injects code into what is typically a web-browser in order to make the application perform in ways it is not supposed to. It is regarded by many experts as the number one vulnerability on the web today. There are three different types of XSS attacks – reflected XSS, stored XSS and DOM XSS.

- **Reflected XSS (Non-Persistent)**
  Reflected XSS attacks are the more common form of XSS attacks. Firstly, the attacker must source a webpage which is vulnerable to the permanent injection of malicious scripts. This can be done by trial and error, with a common method entailing entering JavaScript code with the functionality to open an alert box into a search box in a webpage. When the search is run, if the webpage opens the alert box it means the webpage is vulnerable. The attacker then gets the URL of that vulnerable webpage and adds their malicious JavaScript code to the URL. Through techniques such as phishing, they distribute this URL as a link via emails and posting to forums with the hope an unsuspecting victim will click on the link. When the link containing the malicious JavaScript code is clicked, the web server with the compromised URL is reflected back to the victim, and their browser shows the vulnerable webpage as normal, but also runs the malicious JavaScript code in its URL when parsing the webpage as the webpage has come from a trusted server in context of the victim. This JavaScript code can be used to steal personal information such as cookies containing username passwords and personal data, which are in turn sent back to the attacker. Attackers often attempt to mask the compromised part of the URL to avoid arousing suspicion from the user, such as encoding the ASCII characters in hex.

- **Stored XSS (Persistent)**
  Stored XSS attacks are a more damaging form of cross-site scripting and can destroy the integrity between webpages and their users. Vulnerabilities in webpages are located in the same method undertaken when carrying out a reflected XSS attack. However, instead of targeting search bars, parts of the webpage which take user input and save to the webpage's server are targeted, such as a comment section. Malicious JavaScript code in a HTML tag is entered as a comment and saved to the server. When the page containing the now published comment is accessed in future by other users of the webpage, the JavaScript in the comment is run. Victims can have their login details and the like stolen from cookies, which the code can relay back to the attacker. Stored XSS attacks are difficult to carry out, as it is harder to locate a trafficked website and one with vulnerabilities that allow permanent script embedding.

- **DOM XSS (Document Object Model XSS)**
  DOM based XSS attacks provide a payload which is executed as a result of modifying the DOM environment of a victim's browser. While the HTTP response does not change as it does in reflected and stored XSS attacks, the client-side code executes in unprecedented ways due to the modifications made in the DOM environment. An example of this is, as with reflected and stored XSS attacks, finding a vulnerable webpage. Similar to reflected XSS attacks, the URL is modified, but instead of sending the payload as JavaScript in the URL it only sends script which modifies the DOM of the website it is loaded into. This

modification of the DOM can allow attackers to echo sensitive information onto the webpage such as cookie data.

When developing a website, developers should take ample care to follow a form of what is known as a security development lifecycle, or SDL. This typically involves the four step of preparation, analysis, determine mitigations and validation. The main goal of following an SDL is to reduce the number of security and design flaws in the creation of an application. A key rule to ensure security is to presume that all incoming data coming into an application is form an untrusted source. Checking user input using proper sanitation is the simplest method of defence. This may include setting maximum and minimum lengths of data, regex functions to prevent undesirable data types from being entered and character encoding on special characters such as HTML script tags. Further measures such as expiring a session if two separate IP addresses attempt to access an account on an application at the same time can also be taken. Web Application Firewalls, or WAFs, are also an option. These firewalls attempt to match incoming HTTP requests with a similar request known to be an XSS attack in the firewalls database. If there is a match, the request is blocked. WAFs unfortunately are not the most reliable and even when a match is found and the XSS attack is blocked, all of the webpages traffic must be redirected which gives further issues such as poor latency.

2) SQL Injection attacks involves the manipulation of an SQL query when it is run on a webpage. This can result in the query returning data which it originally did not intend to return or bypass a webpages login without the correct login credentials. SQL injection can also be used to compromise servers and carry out denial of service attacks. Many high profile data breaches in recent years have been the result of SQL injection attacks. The main vulnerability which allows attackers to carry out these attacks is the incorrect sanitation of user input on a webpage.

An example of this is an SQL injection on a login page. If a webpage requires a user to login using a username and a password, it takes both values entered and uses them as parameters in an SQL query. The SQL query would in turn check the database for an entry with an identical match of that username and password, and if such an entry exists a value of true would be returned and the user would be logged in. On the contrary, if the match does not exist a value of false would be returned and the user would be denied access. However, if the webpage does not have the correct measures taken to sanitize its inputs, using SQL injection the query can be modified to return true and allow the user access even with an incorrect username and password. A method of carrying this attack out would be instead of entering a password into the password input, an inverted comma is used followed by an SQL query. The inverted comma is used as this will close the string parameter input if the webpage does not have correct sanitation, and all characters entered after will spill into the webpages original SQL query. In order to return a true value, the SQL query 'OR 1=1—' is then placed after the inverted comma, which always returns true. This now becomes part of the original SQL query, and the double hyphen renders any of the original query after it redundant. The login query will then always return true and allow the attacker access to the webpage without the necessary credentials.

While the above example adheres just to exploiting a simple login function in a vulnerable webpage, SQL injection can be used in complex attacks which include but are not limited to deleting, updating and inserting data, stealing credit cards details and executing trojan horses. When developing a website, developers can take certain precautions to prevent SQL injection attacks from occurring. Similar to preventing cross-site scripting, the security development lifecycle should be followed and all inputs in the application should be correctly sanitized with regular expressions and the like. There are also the options of using prepared statements and stored procedures. Prepared statements create the SQL query first, and all user input is considered as data parameters. Stored procedures encapsulate the SQL queries and consider all user input as data parameters. Both prevent the original SQL query from being modified and also have the added bonus of being much quicker than standard queries.

Question 4

1) A use-after-free memory vulnerability occurs when a pointer is dereferenced while pointing to an object that has already been freed from the heap. The pointer which was pointing to address the memory block the data resided in now becomes invalid and is known as a 'dangling pointer', as control of the memory block has now been passed back to the memory manager. If another object is placed in the memory address of where this pointer is pointing to, it will naturally be pointing to the incorrect data. This can be exploited to change the flow of control and read and change data in a program. This vulnerability can be used to exploit several issues, such as crashing programs using DDOS. It can be prevented through methods such a reference counting.

An example of a use-after-free occurring could be when an object is created in a banking system with a user's details and their total savings in the account, with a pointer pointing to this object. If the object is freed and the pointer is left dangling, the memory is dealt with by the memory manager. Another object can be allocated in its place, and therefore the pointer will be referencing an incorrect object and by placing memory addresses of functions in this object in place of, for example, the savings variable, we can change the control of the flow in the program and can cause that function to execute. Unexpected behaviour can also occur, such as when the memory manager decides to re-assign the freed memory by its own accord.

2) The EMSI vulnerability occurs when an attacker overflows a buffer in order to change the value of a pointer in memory. This can be used to point to a return address of a function which is writing to a particular piece of memory from an input. Essentially, as the attacker has full control of where the value points to, they can pick and choose where in memory the input is stored to, which can result in serious breaches of a program and its data. As this attack uses a pointer which directly points to a piece of memory and not memory on the stack which will travel through the canary word on its way to the desired return address, the Random and Terminator canary mechanisms cannot detect the attack when it is taking place.
The XOR random canary however is effective against the attack, as it also binds the return address to the random canary value. It is an extended version of the Random canary.