Int. J. Advanced Networking and Applications
Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

A Survey of Network-based Security Attacks

Koffka Khan
Department of Computing and Information Technology The University of the West Indies, Trinidad and Tobago, W.I
Email: koffka.khan@gmail.com
Wayne Goodridge
Department of Computing and Information Technology The University of the West Indies, Trinidad and Tobago, W.I
Email: wayne.goodridge@sta.uwi.edu.com

----------**ABSTRACT**----------

Cross Site Scripting, SQL Injection, Denial of Service (DOS), Buffer Overflow and Password Cracking are current network-based security attacks that still looms on the Internet. Though these attacks have been around for decades and there exist protective mechanism for overcoming them they are still relevant today. In this paper we describe the basic workings of these attacks and outline how companies and individuals can mitigate these attacks. By taking the necessary precautions the severity of these attacks can be diminished.

Keywords – **Cross Site Scripting; SQL Injection; Denial of Service (DOS); Buffer Overflow; Password Cracking; security attacks; Internet.**

Date of Submission: Jan 29, 2019 Date of Acceptance: Mar 01, 2019

I. INTRODUCTION

Network-based security has become ever more important with the advent and increasing connectivity brought about by the rapid growth of the Internet. Many devices are now connected, and this brings about the need for increased security against intruders [29].

Many attacks have been around at the start of the Internet. Though many solutions have been proposed they are not implemented properly or not at all. This is due to the cost, time, effort and personnel needed to implement these solutions. Still today many companies and individuals lack the awareness of the threats out there. This means that attacks such as Cross Site Scripting [30], SQL Injection [20], [8], [36], Denial of Service (DOS) [53], [51], Buffer Overflow [43] and Password Cracking [1], [54] are still relevant and viable attacks in the today's world.

This paper aims to outline these attacks and in so doing increase the awareness to readers. It also gives practical examples for easy understanding. Finally, some measures of coping and/or mitigating some of these attacks are given. It is hoped that the reader will gain interest in network security attacks by reading this paper and see the role of security in their own homes and organizations. The reader should be able to understand the need for protection against such attacks outlined.

This work consists of four sections. Section II presents a categorization of network-based attacks. Section III gives details of the attacks defined in the taxonomy. Finally, the conclusion is given in Section IV.

II. CATEGORIZATION OF PRESENT-DAY NETWORK ATTACKS

The network-based security attacks are categorized into (1) Cross Site Scripting, (2) SQL Injection, (3) Denial of Service, (4) Buffer Overflow, and (5) Password Cracking. These categories are shown on Figure 1.

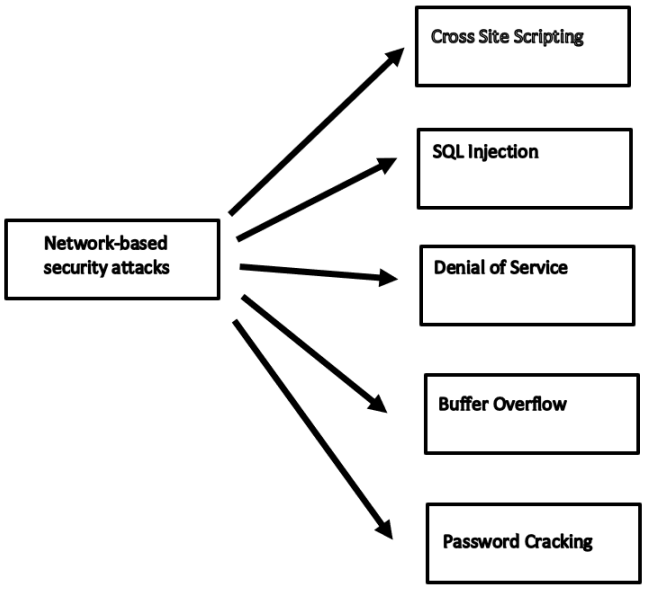

Fig. 1. Current network-based attacks.

III. NETWORK BASED ATTACKS

A. Cross Site Scripting

Cross site scripting (see Figure 2) is the number one vulnerability on the web today. In the early days of the internet Tim Berners-Lee at CERN contemplated on how the web will work. The web use Hypertext Markup Language (HTML) to format and display webpages [17]. An HTML document consists of tags. It starts with a start tag like this, <HTML> and closes with an end tag like this </HTML>. Anything between angle brackets is read as an instruction. For example, to get bold text place a start <B> tag and a close </B> tag. The text in the middle of the start and end tags becomes bold. Thus, the angle brackets, wherever they are in the document, mean "an instruction is coming here." However, if you want to put an angle bracket, which is basically a less-than sign, into your

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

document you do something called escaping. Instead of sending the angle bracket, you send an ampersand (&), and then "lt" for less than, and then a semicolon. This means, when rendered it will become an angle bracket. Therefore, in the old days of the world wide web, you could send a request, and the document would come back, and the angle brackets would not mess everything up.

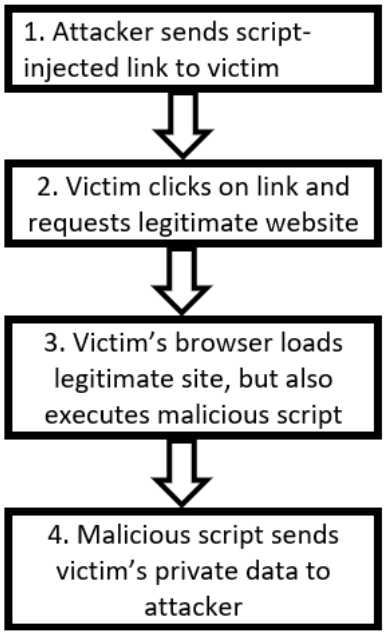

Fig. 2. Cross Site Scripting.

Then the internet started to become more interactive. JavaScript was invented. JavaScript is a programming language that sits in the middle of web pages [10]. You start with a tag in the middle of your document. You start with a <SCRIPT> start tag and a closing </SCRIPT> tag. JavaScript is a separate programming language. For example, you can declare variables and do calculations. Thus, you can create an entire program and that program can affect the document. Output from the program can be placed into the rest of the "markup" text. Therefore, JavaScript is dangerous. It can do anything to the web page. But imagine if you could get JavaScript embedded on a webpage, say, the login page of an online bank. You could tell it that, instead of just taking the username and password and sending them to the bank's servers, first, it should send them to a third party. Further, when this third party got the passwords the user won't know what has happened. The third party could log into the bank. JavaScript is dangerous because it lets you do anything on a web page. So, how do you get it in there? Let us take a Google search bar. Whatever I type in that search bar, "hypertext", will probably appear on the next page a couple of times. However, what happens if, instead, I type in an italic tag? What won't happen is that Google will send the whole page in italics. The Google server have converted the < tag into less-than &lt. Let's imagine that instead of typing "hypertext", I type within the <SCRIPT> </SCRIPT> tags. If the web developer forgets to do that little trick that changes them from less-than signs to that code that means "put a less-than sign in there," the web server puts the page out, and the web browser looks at that and goes, "That is JavaScript code! I'm going to run that!" and it does. Thus, anywhere on your site involving user input is very important. For example, someone sending you their age which you forget to escape, and someone types in a little bit of code there instead, makes your web site completely vulnerable.

B. SQL Injection

SQL injection (see Figure 3) is a way to attack websites via their backend database. Sequel or SQL is a language [11] which allows you talk to databases. It's very human readable. Thus, you can say things like, "SELECT * FROM TABLE" where * means information from all database tables. So, basically you can pretty much type commands in near English into SQL, and you'll get results back from your database. This has existed for years and years and years. It worked fine until the Web came along. Now people are looking at websites and are thinking, "These websites need to be hooked up to databases." In the initial development of the internet, it was pretty much "I'm am going to request a document and you're are going to send that document back to me." However, eventually people worked out that what do you really wanted to do was send a document and have different things come back depending on what you sent. Maybe you could type in a search request, and that would go to a database and pull back something. This is fine.

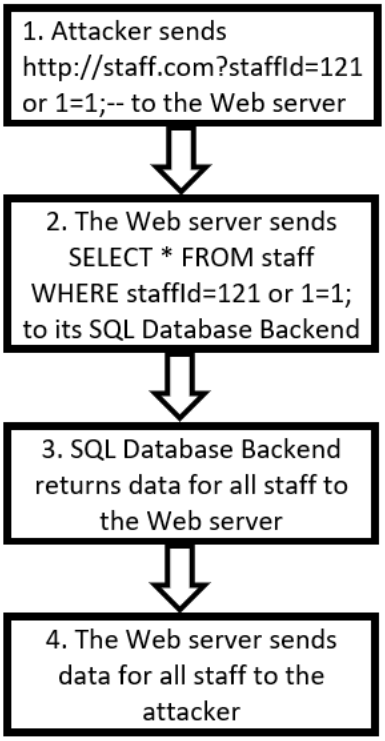

Fig. 3. SQL Injection.

Some programming languages dealt with this in a sensible way, but unfortunately some did not. And one of the most notable ones that didn't is a language called Hypertext Preprocessor (PHP) [42]. PHP makes web

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

programming much more accessible. The trouble is, that if you're not careful, there's a lot of ways to go wrong. And this isn't just PHP, but I'll use it as an example. You talk to a database by issuing a command by typing “John” in a textbox and clicking send. The database would create this command SELECT * FROM users WHERE username equals "John" and this works. The database will send back all the details it knows about the user called "John". But the catch is those quotation marks. Let's say, for example, that I have a web form that lets me login and I type in John, and it sends that and brings back "John". Now if I type in John with a quote mark in it, and if you are not careful, what will happen is the language will send something like this. SELECT * FROM users WHERE username equals "John"" as I had put a quote mark in, and then it put a quote mark in. It fails because the quote marks don't match up. And the whole database crashes and sends back an error message. The big problem is putting in any text that has quote marks. An attacker can do a lot of damage that way because SQL does not just have SELECT statements [23]. It has INSERT to add new elements to the database. It has UPDATE to change elements in the database. It has DELETE to remove elements from the database. If I were to type a username that was John"; and then put another command in there, like, 'DELETE'. So, I would type in the textbox John"; DROP ALL DATABASES. The command would look like SELECT * FROM users WHERE username like John"; DROP ALL DATABASES; The database will go "Well that's exactly what I should do."

It's is going to understand that there's a new command at the semi-colon and that it should delete everything [25]. The main way around it is escaping. When there is dangerous character, like a quote mark, you put a slash before it. You go through, and you use a function that says, "Everywhere there is a quote mark, put this slash before it. And this should occur before you send it to the database." Input comes in from the user, add some slashes to it to make it safe, and send it out to the database [34]. The database will look at those slashes and will go, "Right, every time there's one of those (the backslash), this thing (the quotation marks) is coming next? Just treat it as a regular quote mark. Don't treat it anything special, it's in the text, just treat it as that." However, if you want to send an actual slash, you send two slashes. The first one to say, "Treat the next one as a real character", and then the second is a real slash.

C. Denial of Service

Denial of service attacks have been around for quite some time. For instance, with an internet connection over a 56k modem it is incredibly easy to perform a Denial of Service Attack. In those days particularly if you happen to irritate someone who is on an enormous university connection, at say, 1M. Though this isn't that much bigger by today's standards in those days all it meant was they sent a little message on their system, which sends as much traffic as possible to your system. And if their system is bigger than yours, your internet connection gets saturated, and you can't send anything in and out. At which point you will have to literally hang up the phone to dial in again and get a new IP address so they would not be able to find you.

This was how it worked for a period. Until hackers involved started creating botnets [4], [44]. They started writing viruses that instead of destroying data, would go in and take over other people's internet connections. They would find broadband users, generally in the world, who would be running unsecured versions of Windows XP or 98 for example. They would quietly install their software in the background, and then would use those unsuspecting users' internet connections to launch a big denial of service attack. This was a distributed denial of service (DDOS) [55], [7] so, instead of having one big computer, you had lots of little computers, hundreds, thousands, maybe tens of thousands. All sending as much traffic as they could against one company. And it didn't matter how big that company's internet connection was. Ultimately, ten thousand people all reloading their website or turning out as much traffic as possible as fast as possible is going to take down their network connection. It is used for ransom. It was found that in the 2000s gambling companies, finance companies, and anyone whose job, whose livelihood, depended on being up and online all the time 24/7, was being held for ransom. They would get a call, an email, or a message that said, "If you don't pay us an amount your website's is going to go down for quite a while." There are defense strategies. You can generally hire a very expensive company to try and mitigate this, at which point it does become a bit of a bit of a protection racket. But ultimately Microsoft got their act together and the number of zombie computers, as they were called, started to decrease. Also, the internet started getting more and more and more and more bandwidth. Thus, you could hire a net connection that could stand up to reasonable denial of service attacks for not too much. Stacheldraht [12] is a classic example of a DDoS tool, cf. Figure 4. It utilizes a layered structure where the attacker uses a client program to connect to handlers, which are compromised systems that issue commands to the zombie agents [46], which in turn facilitate the DDoS attack. Agents are compromised via the handlers by the attacker, using automated routines to exploit vulnerabilities in programs that accept remote connections running on the targeted remote hosts. Each handler can control up to a thousand agents.

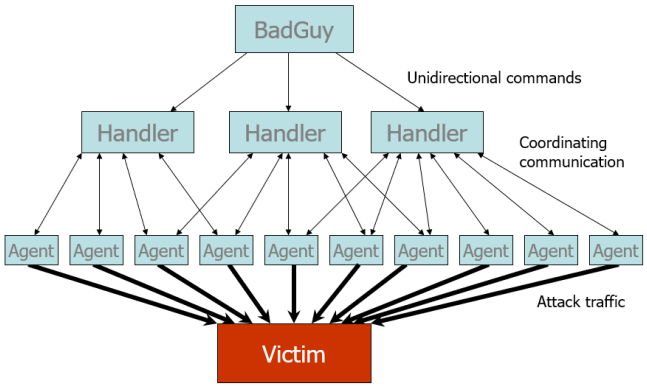

Fig. 4. Distributed DOS.

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

Now the new threat is something called Amplified Denial of Service (ADOS) [2]. And it's not a new threat as such, it's just a new common threat that's been theorized about for a while. It is a combination of a couple of vulnerabilities in how some very old parts of the internet work. First let us explore the difference between TCP [14] and UDP [37]. TCP is how most of the web works. It's how the webpage that you're viewing gets sent back and forth. It is a two-way protocol and there's a handshake involved. You request something, and then that request is acknowledged, and you get something back and as all the packets go back and forth. Thus, there is two-way conversation going on making sure that everything's arrived in the right order, intact. This means you can use it for webpages and use it for financial transactions on your online bank. You can use it for anything where getting everything through “bit perfect” is required. UDP is very much opposite of that. UDP sends the stream of data. The two-way conversation does not exist. This is what you use for voice over IP (VoIP) [16] [48]. It does not matter if a bit of it gets lost or a bit arrives in the wrong order. UDP does not have to acknowledge data sent and say "Yes! I approve this stream being sent to me." It just kind of arrives and there's not much you can do about it. There is a flaw in the UDP protocol, or at least in some implementations of it as you can essentially spoof the return address. My computer can claim that I am someone else entirely. This would not normally be a problem because most well-designed network protocols will only let you send on a small amount of data. I send a small request to them. They send a small request onwards etc… And it's not really a problem.

Except, there is something called the Network Time Protocol (NTP) [31], [32], [33]. The Network Time Protocol keeps all the clocks in your phone and your laptop in sync to almost to the millisecond. The problem lies with the command: "MONLIST". It sends the details of the last 600 people who requested the time from that computer. So, when I send a tiny request (send time information 206 times), using the MONLIST command to the time servers (all time servers are on enormous connections), spoof where it came from, and they will send an enormous amount of data 206 times the amount of data to that poor computer with the spoofed address. This is NTP amplification, but it's not the only amplification attack. There's been DNS [47] for a while, there are a couple of others that security researchers are hinting at, but don't want to release the details. Recently, we have seen one, maybe two, terabit per second attacks. That is a hundred thousand times more than your broadband connection. It's something that is on the scale of disrupting the entire Internet, rather than just disrupting one computer. How can you defend against it? Well... you can't. I mean you can hire a company that claims to be able to block a lot of attacks and they can, work at the network level to try and filter it all out. But ultimately, against an attack of that size there's not much a victim can do. But what you can do is, campaign to get the relays, which forces the amplification vectors to shut down. The relays do this by blocking, filtering traffic and then shutting down.

D. Buffer Overflow

A buffer overflow exploit is a situation where an attacker is using some, probably low-level C function or procedure [40] to write a string or some other variable into a piece of memory that is only a certain length. However, the attacker is trying to write something in that's longer and then overwrites the later memory addresses, and that can cause all kinds of problems.

The first thing we should talk about, probably, is roughly what happens in memory with a program when it's run. Now, let us use C programs in Linux [27]. But this will apply to many different languages and many different operating systems. So, when a program is run by the operating system (so the attacker is in some shell and types in a command to run a program) the operating system will effectively call, as a function, the main method of the code the program is running on. But your actual process, your executable, will be held in memory in a very specific way. This is consistent between different processes. So, the attacker has access to a big block of RAM (see Figure 5). We don't know how big our RAM is because it can be varied, but we use something called Virtual Memory Address Translation to say that everything in one end of the RAM memory, this is 0. 0x000… the “base” of the memory. And the other end (the “top”) is 0xFFF. So, this is the equivalent of "11111111" memory address all the way up to 32 or 64 bits.

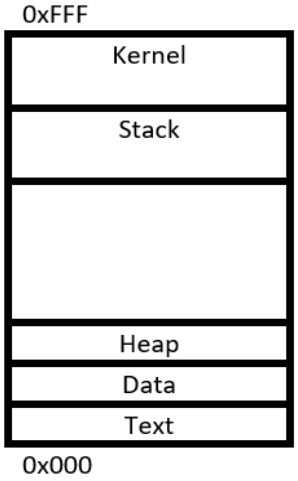

Fig. 5. RAM memory.

Now, there are certain areas of this memory that are always allocated to certain computational elements. So, at the top we have kernel computational elements. So, this will be command line parameters that we can pass to our program and environment variables etc... In the lower portions we have something called the text. That's the actual code of our program. The machine instructions [38] that we've compiled get loaded in there. Now that's read-only, because we don't want to be messing about down there. Even lower we have data. So, uninitialized and initialized variables get held here. And then we have the heap. It's where you allocate large things in your memory.

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

What you do with the heap is up to your program. Even lower, and perhaps the most important bit, in some ways anyway, is the stack. The stack holds the local variables for each of your functions and when you call a new function like, let's issue "printf" and then some parameters that gets put on the end of the stack. So, the heap grows in a downward direction as you add memory, and the stack grows in an upward direction. We'll just focus on the stack, because that's where a lot of these buffer overflows happen. You can have overflows in other areas, but we're not going to be dealing with them in this paper. At the upper end of the stack we have the high memory addresses (0xFFF...) and 0x000 at the lower end. As the stack grows upwards, so when we add something onto the end of the stack it gets put on this side and moves in a upward direction. Recall the attacker has some program that's calling a function. A function is some area of code that does something and then returns to where it was before. When the calling function wants to make use of something, it adds its parameters that it's passing onto the stack. So, let us assume parameter A and parameter B (see Figure 6), is added into the stack in reverse order.

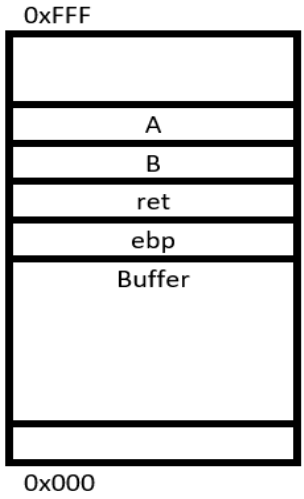

Fig. 6. Stack.

The Assembler [38] code for this function will make something called a "call" and that will jump to somewhere else in memory and work with these two parameters. It's the nature of this stack that causes problems. Let's look at some code and then we'll see how it works. So, it's a piece of C code (see Figure 7). It's a very simple C code that allocates some memory on the stack and then copies a string into it from the command line.

```
#include <stdio.h>
#include <string.h>

int main (int argc, char** argv)
{
        char buffer[500];
        strcpy(buffer, argv[1]);

        return 0;
}
```

Fig. 7. C program.

So, we've got the main function for C that takes the number of parameters given and a pointer to those variables. They'll be held in kernel area of our memory. We've allocated a buffer that's 500 characters long and then we call a function called "string copy" (strcopy) which will copy our command line parameter from argv into our buffer. Our function puts on a return address which is replacing the code we need to go back to once we've done strcopy. So that's how main knows where to go after it's finished. Then we put on a reference to the base pointer in our previous function. We won't worry about that too much because it's not relevant particularly to this paper. This is just going to be our EBP base pointer (EBP is a pointer to the top of the stack when the function is first called). This is our allocated space for our buffer, and it's 500 long. If we write into it something that's longer than 500, we're going to go straight past the buffer, over this, and crucially over our return variable. That's where we point back to something we shouldn't be doing. You can walk through the following code and then see if it works.

You can use a Kali Linux distribution, which has all kinds of slightly dubious password cracking tools and other penetration testing tools. It's meant for ethical hacking. Run the small function that does our copy from the command line. Run your vulnerable code with "Hello". This will copy "Hello" into this buffer and then simply return, so nothing happens. Now we're going to run something called GDB, which is the Linux command line debugger. Type in "list" and it shows us the code for our function. So, we can see it's just a compiled function. It knows this because the compiler included this information along with the executable. We can also show the machine code for this so we can type "disas main" and we can see the code for "main()". This line here, sub of 0x1f4 from %esp, that's allocating the 500 for the buffer. That is, we go 500 in the upward direction and that's where our buffer goes. So, buffer's sitting to the top in Fig. xxx but it is lower in memory than the rest of our variables. We can run this program from GDB and if it crashes, we can look at the registers and find out what's happened.

We type "run Hello" and it will start the program and say "Hello". And it's exited normally. Now, we can pass something in a little bit longer than "Hello". If we pass something that's over 500, then this buffer will go over this base pointer and this return value and break the code. The program crashes. Let us print the "a" character" 506 times and see what happens. Just a little bit more than 500 so it's going to cause somewhat of a problem but not a catastrophe. Run the program. A segmentation fault occurs [5], [9]. Now a segmentation fault is what a CPU will send back to you when you're trying to access something in memory you shouldn't be doing. Now that's not what happened because we overwrote somewhere, we shouldn't; what has happened is the return address was half overwritten. For example, there is nothing in memory at 0xb7004141, and if there is, it doesn't belong to this process. It's not allowed, so it gets a segmentation fault. So, if we change this to 508, we're going two bytes further along, which means we're now overwriting the entirety of our return address. We're overwriting this "ret" here with

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

41s. Now if there were some virus code at 414141, that's a big problem. So that's where we're going with this. So, we run this, and you can see the return address is now 0x414141. I can show you the registers and you can see that the construction pointer is now trying to point to 0x414141. This means that it's read this return value and tried to return to that place in the code and run it, and of course it can't.

Let us change this return value to somewhere where we've got some payload we're trying to produce. Now in fact this payload is just a simple, very short program in Assembler, that puts some variables on the stack and then executes a system call to tell it to run a shell to run a new command line. If I show this code, our shell code, this code will depend on the Linux operating system and whether you're using an Intel CPU or something else. This is just a string of different commands. Crucially, this xcd / x80 is throwing a system interrupt, which means that it's going to run the system call. That's all we're going to do about this. What this will actually do is run something called ZSH, which is an old shell that doesn't have a lot of protections involved. Let's go back to our debugger. We're going to run again but this time we're going to run a slightly more malicious piece of code. We're going to put in our \x41s times by 508 - and then we're going to put in our shell code. So now we're doing all 41s and then a bunch of malicious code. Finally, the last thing we want to add in is our return address, which we'll customize in a moment. To craft an exploit from this, what we need to do is remember the fact that strcopy is going to copy into our buffer. So, we're going to start here. We want to overwrite the memory of this return address with somewhere pointing to our malicious code. Now, we can't necessarily know for sure where our malicious code might be stored elsewhere on the disc, so we don't worry about that or memory. We want to put it in this buffer. So, we're going to put some malicious code and then we're going to have a return address that points back into it. Memory moves around slightly. When you run these programs, things change slightly, environment variables are added and removed, things move around. So, we want to try and hedge our bets and get the rough area that this will go in. In here, we put \x90. That is a machine instruction for "just move to the next one". Anywhere we land in that No-Op is going to tick along to our malicious code. So, we have a load of \x90s here... then we have our shell code.

That's our malicious payload that runs our shell. Then we have the return address, right in the right place, that points back right smack in the middle of these \x90s. What that means is, even if these move a bit, it'll still work. It's like having a slope. Anywhere where we land in here is going to cause a real problem for the computer. We need to put in some \x90s, we need to put in our shell code, which I've already got, and we need to put in our return address. If we go back to the code: we change the first \x41s that we were putting in, and we change to 90. We're putting in a load of No-Op operations. Then we've got our shell code and then we've got what will eventually be our return address. And we'll put in 10 of those because it's just to have a little bit of padding between our shell code and our stack that's moving about. So, if we write 508 bytes, it goes exactly where we want: over our return address. But we've now got 43 bytes of shell code and we've got 40 bytes of return address. We'll change this 508 to 425, and so now this exploit here that we're looking at is exactly what I hoped it would be here. Some \x90 no operation sleds, the shell code and then we've got our return address, which is 10 times four bytes. We run this and we've got a segmentation fault, which is exactly what we hoped we'd get.

E. Password Cracking

Bad passwords is a real problem. It's a problem because People like LinkedIn [45] and TalkTalk [6] get hacked, and a bunch of hashed passwords go out onto the Internet. Then within hours' half of them have been cracked. And then people are going: "Oh well this user name and this password's been cracked. Well let's just go and log on over there and see if that username and password combination gets me into their Amazon. Oh! it does? That's good news." And, and so on. Password cracking has massive implications for password security. Hashing algorithms [3], [15] have become longer because they don't hold up as well as the older ones. We don't store passwords unencrypted in a database because that's a terrible idea. What we do is we pass them through something called a "One Way Pseudorandom Function" [28], [22], [35], [21]. Which basically take some plain text password and turns it into gibberish. And then, when someone tries to login, we do the same operation on what they just typed, and if the gibberish matches, we know they've taught in their password correctly, without actually having to know what their password is. But if these hashes get dumped on the internet then we can't reverse them because they're just random nonsense but what we can do is test the load of different words by hashing them and seeing if the hashes match any of the ones in the dictionary and if they do, we know we've cracked their password. This is easy to do. I'm going to show you it and it's got me scared me the first time.

Hashcat [19], [26], [39] is one of the foremost password cracking tools. It lets you do lots of different types of password cracking which I'll talk about and it does it very quickly because it makes use of the graphics card or graphics cards in parallel. A present-day graphics card is capable of around 10 billion hashes per second. It takes 40 billion plaintext password hypotheses, hashes them using MD5, and compares them to a list at a rate of 40 billion per second. Hashcat is run off the command line. There is a file with a list of hashes that comes with Hashcat. There's about six or so thousand hashes in it that range in difficulty. So, some of them are going to be "password1" because that's what some people's passwords are, and some of them are going to be much longer, so 20 or 30 characters, almost random, and they're going to be very difficult to crack. MD5 produces a hundred- and twenty-eight-bit hash [41], [49]. The problem is that lower standard hashes like MD5 and SHA-1 [50] still get used a lot for back end storage. Change your hashes to something like SHA-512 [18], [24]quickly, because this is not

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

acceptable. Hashing takes longer for the GPU to process and so you will go down from 40 billion to, you know, a few million or a few thousand for good hashing that's been iterated a lot of times. This makes the process insurmountably harder.

As a user, it just means you must have a password that's acceptable, but you have to, in a way, assume that some of the websites that you use won't know what they're doing and will have it stored in MD5. If it's still in plain text, then all bets are off, there's nothing we can do. Okay, right, so let's just run this in brute force mode. So, the first type of password cracking, which sees some use but not a lot, is brute force. So, this is simply a case of starting with "AAAAAAA" and then "AAAAAAB" and "AAAAC" and so on for different character sets. If we assume that it's going to be some subset of passwords that use only lowercase letters, we can brute force those very quickly, especially if they're not very long. So, what I'm going to do first is I'm going to run an attack on these passwords of, let's say, seven-character passwords all with lower case letters. Hashcat attack mode 3, which is brute force, example0.hash (the hash file) and then my mask which tells me what character sets I'm going to use. So, L is a lowercase letter, so 1, 2, 3, 4, 5, 6, 7 lower case letters. Run your code. Okay, not very many, because there aren't very many, luckily for these users, lowercase only passwords. With lowercase letters only, there are 26 lowercase characters, 26, to the power of 7, for when we were trying 7 passwords and then for, let's say, six-character passwords with two digits on the end it's going to be 26 to the power of 6 multiplied by 10 to the power of 2. Well, if you're using lower and uppercase, it's going to be (26*2)^7.

If your password is six characters long, it's being cracked right now, and it's being cracked quickly because we can go through all the 6-character passwords in a fraction of a second. For longer passwords, we must make some assumptions about the way that people choose passwords. So, obviously the password "password1" is nine characters, in which brute force is pretty good, but it's not good because it's the number one password to be used. And so on the top of your list of hypothetical passwords, it should be right at the top and the first one you try. This is what a dictionary attack does. We have a dictionary of a list of commonly used words or commonly used passwords, and then we try those. And then we manipulate them slightly, with rules, and we try them again and we append them to other words and try them again and we do lots of different combinations of things and try them again. It's much more effective than brute force, and so it's currently very popular. The hashing rate goes down a bit because you're loading dictionaries and doing word manipulations but it's still quick.

So, let's show you an example dictionary. This dictionary has common passwords that have been cracked from other sources. There are other password lists, like the RockYou list [52] and soon the LinkedIn list, I'm sure, which will have a big impact because they are real passwords of people are using, so if you make a word list out of those passwords that's going to be effective. Use Hashcat, but this time we're going to run in attack mode 0, which is straight dictionary attack. In a big database, you're going to have a lot of people who have "password" and "password1234" and "12341234" and so on. All those people are going to be found this way but what we really want to do is mix up the dictionary little bit, swap a few letters around. So, there are rules that do obvious things like they replace "I" with the number 1. Or they replace "E" with a 3. Or put an "@" in instead of an "&" or something. Toggling case up and down, you know, if a password's viable, then the same password with the first letter as uppercase also probably viable. So, with some luck, we've done a bit of brute force, we've done a basic dictionary attack, we have a few rules just to mix it up, and we've got some passwords.

So how can we get even better? Well, we use a better dictionary. That's the key. This example dictionary is fine, it's not very long, you know some passwords are going to be in it, but as you remember we ran it and it didn't find many passwords. It found some when we ran it through some rules, but it didn't find a lot. So what we really want to do is find a list of actual passwords that people are using in real life and use that. Now, these leaks happen all the time and so passwords are just being dumped out onto the internet all the time. So, there's this password list called RockYou, which is a bit of a game changer in password cracking, in that it has around 14 million or so passwords, actually leaked from a proper database of real passwords that people were using. It was a gaming service or something like this and then it got leaked. And the point is that if you run the RockYou database over these hashes you start to really get results, because there's just much more interesting passwords in the RockYou database, there's just many more of them.

We tried by brute force or by normal dictionary but this RockYou database has changed everything in the sense that it's just so varied that you just get password that you just get passwords that you think are good. So, for readers, you got to think how good are your passwords? Are your passwords better than half the people in the RockYou list, right? And if they aren't, that's probably the next thing you should do, is change them.

IV. CONCLUSION

Cross Site Scripting, SQL Injection, Denial of Service (DOS), Buffer Overflow and Password Cracking are current network-based security attacks that still looms on the Internet. Though these attacks have been around for decades and there exist protective mechanism for overcoming them they are still relevant today. This paper described the basic workings of these attacks and outlined how companies and individuals can mitigate these attacks. By taking the necessary precautions the severity of these attacks can be diminished.

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

REFERENCES

[1] Aggarwal, Sudhir, Shiva Houshmand, and Matt Weir. "New Technologies in Password Cracking Techniques." In Cyber Security: Power and Technology, pp. 179-198. Springer, Cham, 2018.

[2] Ambrosin, Moreno, Mauro Conti, Fabio De Gaspari, and Nishanth Devarajan. "Amplified distributed denial of service attack in software defined networking." In New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference on, pp. 1-4. IEEE, 2016.

[3] Andoni, Alexandr, and Piotr Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions." In Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on, pp. 459-468. IEEE, 2006.

[4] Barford, Paul, and Vinod Yegneswaran. "An inside look at botnets." In Malware detection, pp. 171-191. Springer, Boston, MA, 2007.

[5] Beck, Leland L. System software: an introduction to systems programming. Addison-Wesley, 1997.

[6] Bouwman, Peter, and Hans de Bruin. "Talktalk." In Object-oriented and mixed programming paradigms, pp. 125-141. Springer, Berlin, Heidelberg, 1996.

[7] Chang, Rocky KC. "Defending against flooding-based distributed denial-of-service attacks: a tutorial." IEEE communications magazine 40, no. 10 (2002): 42-51.

[8] Clarke-Salt, Justin. SQL injection attacks and defense. Elsevier, 2009.

[9] Cohen, David M., Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. "The combinatorial design approach to automatic test generation." IEEE software 13, no. 5 (1996): 83-88.

[10] Crockford, Douglas. The application/json media type for javascript object notation (json). No. RFC 4627. 2006.

[11] Date, Chris J., and Hugh Darwen. A Guide To Sql Standard. Vol. 3. Reading, MA: Addison-Wesley, 1997.

[12] Dittrich, David. "The 'stacheldraht' distributed denial of service attack tool." (1999).

[13] Flanagan, David. JavaScript: the definitive guide. " O'Reilly Media, Inc.", 2006.

[14] Forouzan, Behrouz A., and Sophia Chung Fegan. TCP/IP protocol suite. McGraw-Hill Higher Education, 2002.

[15] Frakes, William Bruce, and Ricardo Baeza-Yates, eds. Information retrieval: Data structures & algorithms. Vol. 331. Englewood Cliffs, NJ: prentice Hall, 1992.

[16] Goode, Bur. "Voice over internet protocol (VoIP)." Proceedings of the IEEE 90, no. 9 (2002): 1495-1517.

[17] Graham, Ian S. The HTML sourcebook. John Wiley & Sons, Inc., 1995.

[18] Grembowski, Tim, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott. "Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512." In International Conference on Information Security, pp. 75-89. Springer, Berlin, Heidelberg, 2002.

[19] Guidorizzi, Richard P. "Security: active authentication." IT Professional 15, no. 4 (2013): 4-7.

[20] Halfond, William G., Jeremy Viegas, and Alessandro Orso. "A classification of SQL-injection attacks and countermeasures." In Proceedings of the IEEE International Symposium on Secure Software Engineering, vol. 1, pp. 13-15. IEEE, 2006.

[21] Håstad, Johan, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. "A pseudorandom generator from any one-way function." SIAM Journal on Computing 28, no. 4 (1999): 1364-1396.

[22] Jarecki, Stanisław, and Xiaomin Liu. "Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection." In Theory of Cryptography Conference, pp. 577-594. Springer, Berlin, Heidelberg, 2009.

[23] Jarke, Matthias, and Jurgen Koch. "Query optimization in database systems." ACM Computing surveys (CsUR) 16, no. 2 (1984): 111-152.

[24] Khovratovich, Dmitry, Christian Rechberger, and Alexandra Savelieva. "Bicliques for preimages: attacks on Skein-512 and the SHA-2 family." In Fast Software Encryption, pp. 244-263. Springer, Berlin, Heidelberg, 2012.

[25] Liu, Zhi-Ying, and Cheng-Rong Zhu. "Comparison of approaches to data security implementation based on PHP [J]." Computer Engineering and Design 19, no. 011 (2009).

[26] Llewellyn-Jones, David, and Graham Rymer. "Cracking PwdHash: A Brute-force Attack on Client-side Password Hashing." In Proceeding of 11th International Conference on Passwords (Passwords16 Bochum). 2016.

[27] Love, Robert. Linux Kernel Development (Novell Press). Novell Press, 2005.

[28] Luby, Michael, and Charles Rackoff. "How to construct pseudorandom permutations from pseudorandom functions." SIAM Journal on Computing 17, no. 2 (1988): 373-386.

[29] Madhura, P. M., Palash Jain, and Harini Shankar. "NFC-Based Secure Mobile Healthcare System." International Journal of Advanced Networking & Applications (IJANA) (2014): 0975-0282.

[30] Mereani, Fawaz A., and Jacob M. Howe. "Detecting Cross-Site Scripting Attacks Using Machine Learning." In International Conference on Advanced Machine Learning Technologies and Applications, pp. 200-210. Springer, Cham, 2018.

[31] Mills, David L. "Internet time synchronization: the network time protocol." IEEE Transactions on communications 39, no. 10 (1991): 1482-1493.

[32] Mills, David. Network Time Protocol (Version 3) specification, implementation and analysis. No. RFC 1305. 1992.

[33] Mills, David. Network time protocol. RFC 958, M/A-COM Linkabit, 1985.

[34] Özsu, M. Tamer, and Patrick Valduriez. Principles of distributed database systems. Springer Science & Business Media, 2011.

Volume: 10 Issue: 05 Pages: 3981-3989 (2019) ISSN: 0975-0290

[35] Patarin, Jacques. "How to construct pseudorandom and super pseudorandom permutations from one single pseudorandom function." In Workshop on the Theory and Application of of Cryptographic Techniques, pp. 256-266. Springer, Berlin, Heidelberg, 1992.
[36] Pollack, Edward. "Protecting Against SQL Injection." In Dynamic SQL, pp. 31-60. Apress, Berkeley, CA, 2019.
[37] Postel, Jon. User datagram protocol. No. RFC 768. 1980.
[38] Ramsey, Norman, and Mary F. Fernández. "Specifying representations of machine instructions." ACM Transactions on Programming Languages and Systems (TOPLAS) 19, no. 3 (1997): 492-524.
[39] Ratna, Anak Agung Putri, Prima Dewi Purnamasari, Ahmad Shaugi, and Muhammad Salman. "Analysis and comparison of MD5 and SHA-1 algorithm implementation in Simple-O authentication based security system." In QiR (Quality in Research), 2013 International Conference on, pp. 99-104. IEEE, 2013.
[40] Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk. The C programming language. Englewood Cliffs: Prentice Hall, 1988.
[41] Rivest, Ronald. The MD5 message-digest algorithm. No. RFC 1321. 1992.
[42] Royappa, Andrew V. "The PHP web application server." Journal of Computing Sciences in Colleges 15, no. 3 (2000): 201-211.
[43] Sah, Love Kumar, Sheikh Ariful Islam, and Srinivas Katkoori. "An Efficient Hardware-Oriented Runtime Approach for Stack-based Software Buffer Overflow Attacks." In 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST), pp. 1-6. IEEE, 2018.
[44] Silva, Sérgio SC, Rodrigo MP Silva, Raquel CG Pinto, and Ronaldo M. Salles. "Botnets: A survey." Computer Networks 57, no. 2 (2013): 378-403.
[45] Skeels, Meredith M., and Jonathan Grudin. "When social networks cross boundaries: a case study of workplace use of facebook and linkedin." In Proceedings of the ACM 2009 international conference on Supporting group work, pp. 95-104. ACM, 2009.
[46] Sterne, Dan, Kelly Djahandari, Ravindra Balupari, William La Cholter, Bill Babson, Brett Wilson, Priya Narasimhan, Andrew Purtell, Dan Schnackenberg, and Scott Linden. "Active network based DDoS defense." In DARPA Active NEtworks Conference and Exposition, 2002. Proceedings, pp. 193-203. IEEE, 2002.
[47] Swildens, Eric Sven-Johan, and Richard David Day. "Domain name resolution using a distributed DNS network." U.S. Patent 7,725,602, issued May 25, 2010.
[48] Thirunavukkarasu, E. S., and E. Karthikeyan. "A Security Analysis in VoIP Using Hierarchical Threshold Secret Sharing." In Proceedings of the UGC Sponsored National Conference on Advanced Networking and Applications. 2015.
[49] Wang, Xiaoyun, and Hongbo Yu. "How to break MD5 and other hash functions." In Annual international conference on the theory and applications of cryptographic techniques, pp. 19-35. Springer, Berlin, Heidelberg, 2005.
[50] Wang, Xiaoyun, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1." In Annual international cryptology conference, pp. 17-36. Springer, Berlin, Heidelberg, 2005.
[51] Wankhede, Sonali B. "Study of Network-Based DoS Attacks." In Nanoelectronics, Circuits and Communication Systems, pp. 611-616. Springer, Singapore, 2019.
[52] Weir, Matt, Sudhir Aggarwal, Michael Collins, and Henry Stern. "Testing metrics for password creation policies by attacking large sets of revealed passwords." In Proceedings of the 17th ACM conference on Computer and communications security, pp. 162-175. ACM, 2010.
[53] Yuan, Yuan, Huanhuan Yuan, Daniel WC Ho, and Lei Guo. "Resilient control of wireless networked control system under denial-of-service attacks: a cross-layer design approach." IEEE transactions on cybernetics (2018).
[54] Zaheer, Zainab, Aysha Khan, M. Sarosh Umar, and Muneeb Hasan Khan. "One-Tip Secure: Next-Gen of Text-Based Password." In Information and Communication Technology for Competitive Strategies, pp. 235-243. Springer, Singapore, 2019.
[55] Zargar, Saman Taghavi, James Joshi, and David Tipper. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks." IEEE communications surveys & tutorials 15, no. 4 (2013): 2046-2069.

Author Profile:

Koffka Khan received the M.Sc., and M.Phil. degrees from the University of the West Indies. He is currently a PhD student and has up-to-date, published numerous papers in journals & proceedings of international repute. His research areas are computational intelligence, routing protocols, wireless communications, information security and adaptive streaming controllers.

Wayne Goodridge is a Lecturer in the Department of Computing and Information Technology, The University of the West Indies, St. Augustine. He did is PhD at Dalhousie University and his research interest includes computer communications and security.