SANS

www.sans.org

SECURITY 642

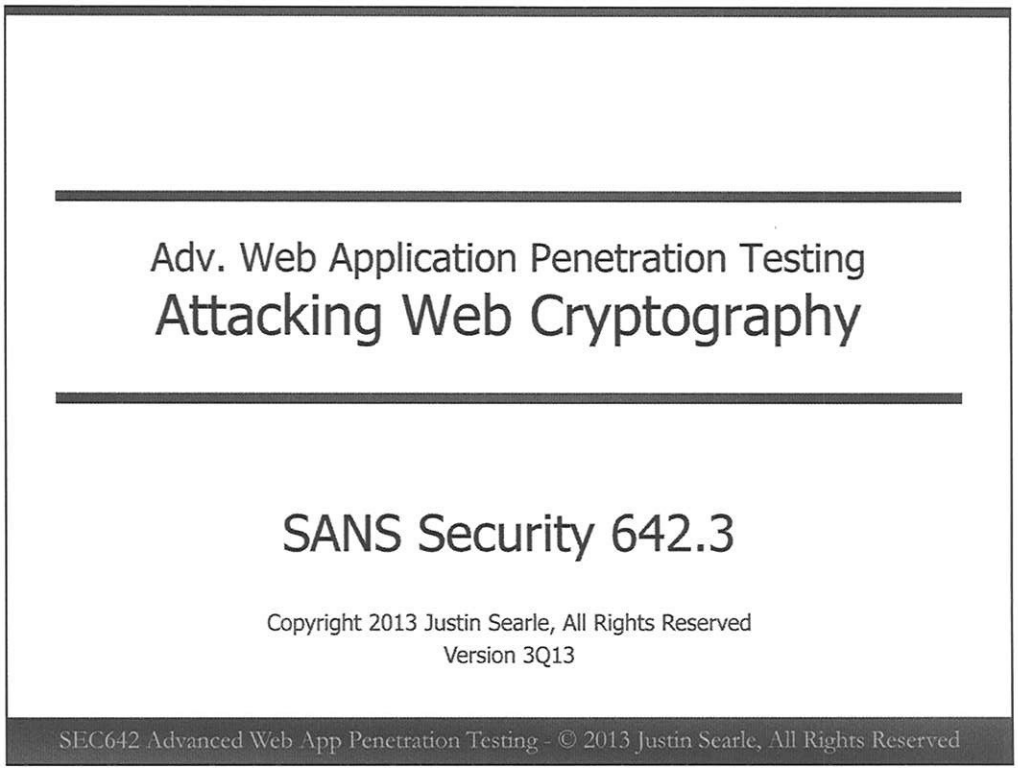
ADVANCED WEB APP PENETRATION TESTING AND ETHICAL HACKING

642.3

Web Application Encryption

The right security training for your staff, at the right time, in the right location.

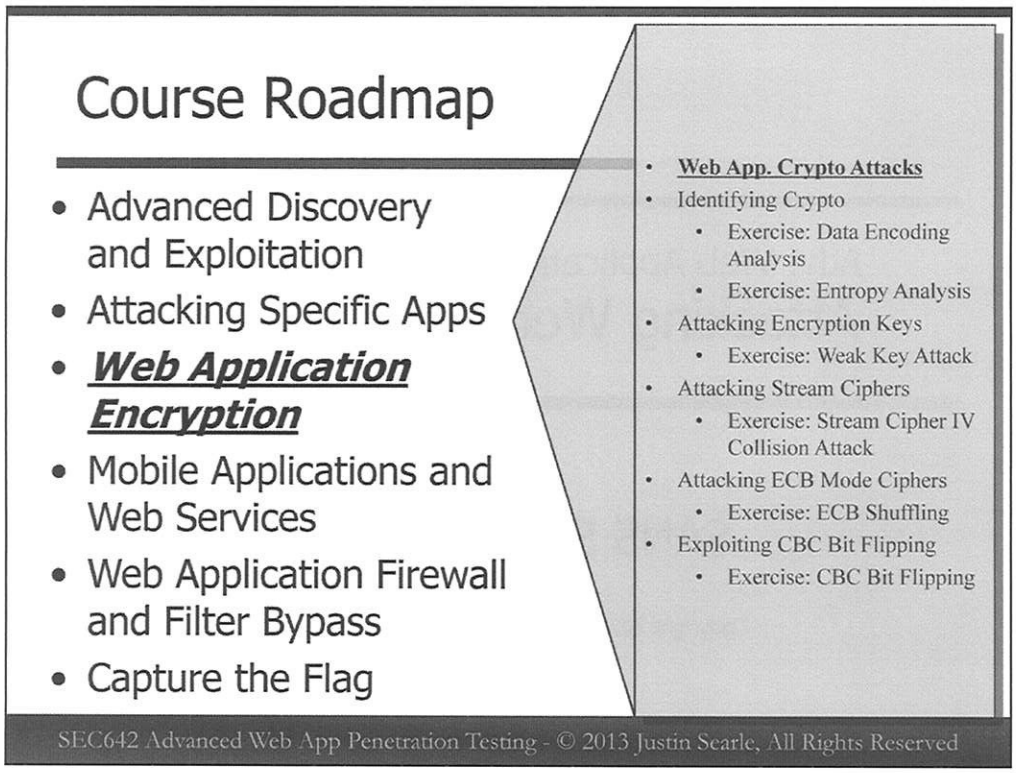
V2013_0902



Welcome to Advanced Web Application Penetration Testing Part 2, Attacking Web Cryptography. In this material we'll look at advanced web penetration testing techniques focusing on exploiting weaknesses in cryptography systems.

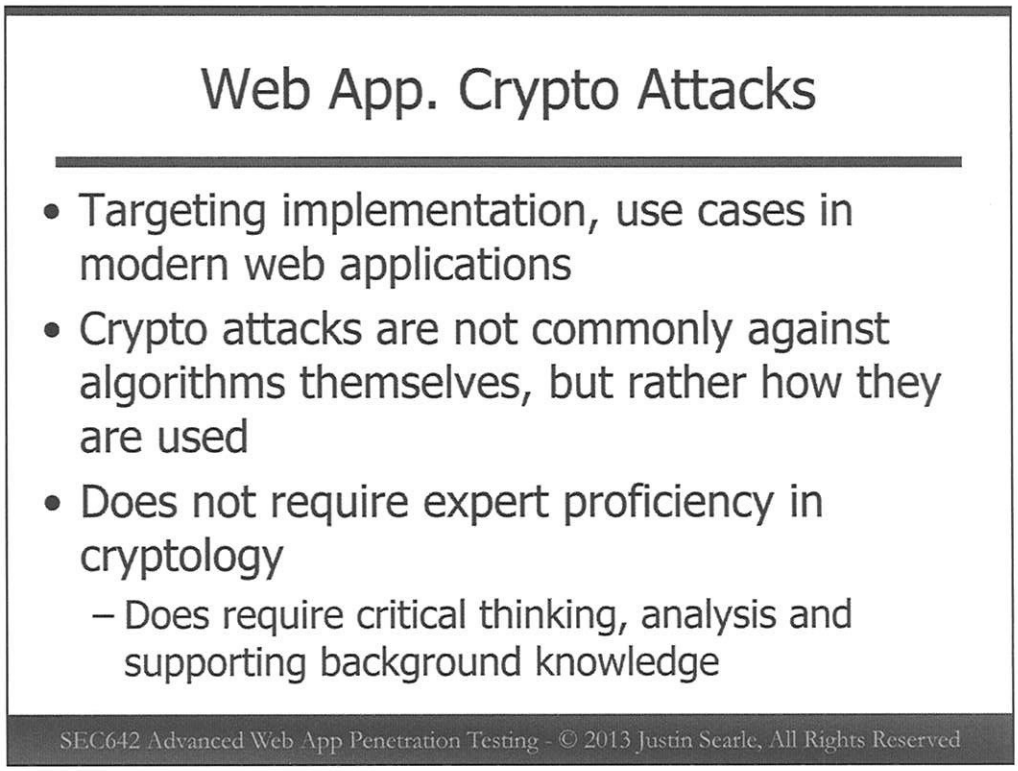
As the author of this material, I encourage you to reach out to me with any questions, comments or concerns. Thank you!

Joshua Wright
jwright@willhackforsushi.com



Course Roadmap

Our roadmap today will take us through multiple areas of attacking web application cryptography systems. We'll begin with a brief introduction of our focus area and what we'll be covering in this section's material.

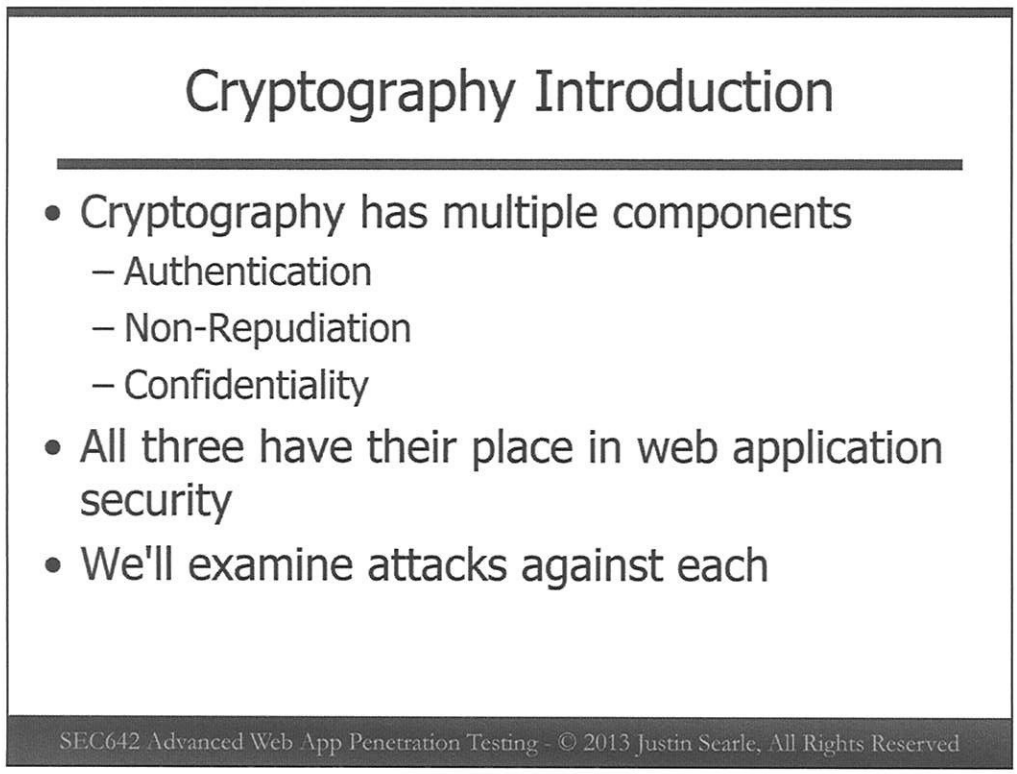


Web Application Crypto Attacks

In my role as a professional penetration tester I've researched and leveraged cryptographic flaws against a variety of systems including web applications. Unfortunately, cryptographic weaknesses are a common area where flaws are present, yet few penetration testers have the skill to investigate, attack and exploit these flaws.

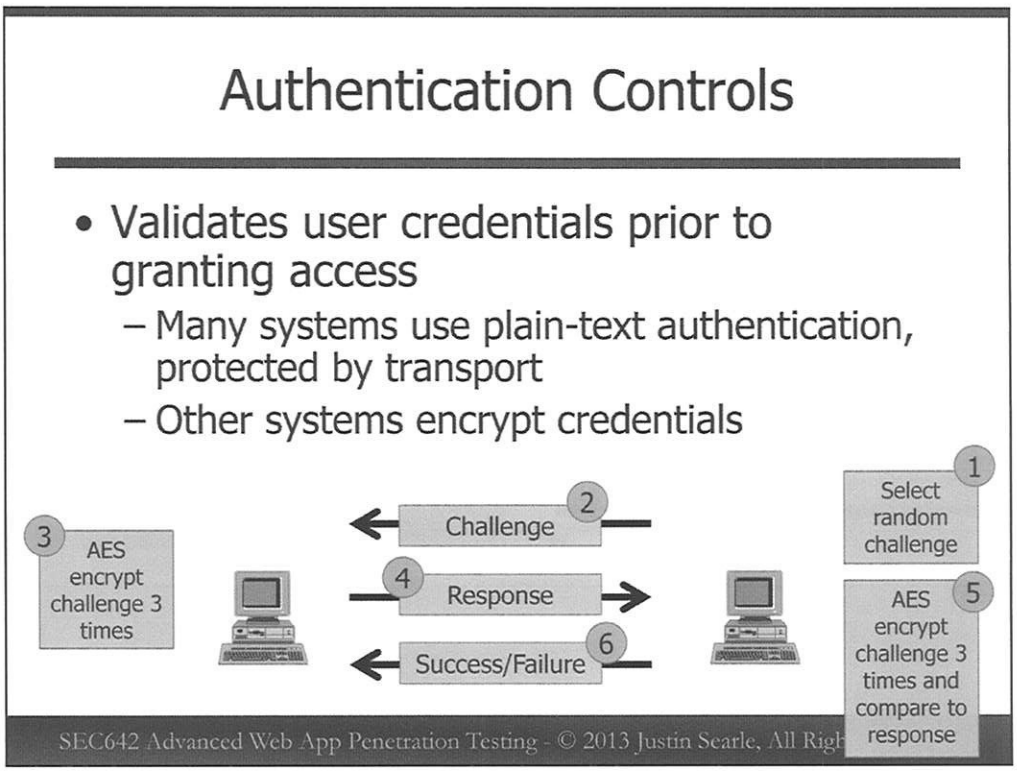
When we investigate web application crypto attacks, we typically target the implementation and use of cryptography in modern web applications. Many popular web programming languages or development frameworks make encryption services available to the developer, but do not inherently protect encrypted data from being attacked, or permit the developer to use cryptography in a weak manner. These implementation mistakes are going to be our focus in this section, as opposed to the exploitation of deficiencies in the cryptographic algorithms themselves.

The upside of attacking implementations is that we do not require an expert-level proficiency in cryptology, the study of cryptography and cryptanalysis. It is not necessary to have a master's degree in math to understand the common weaknesses in cryptography implementations, though it will require critical thinking, analysis and supporting background knowledge on how common cryptography systems function.



Cryptography Introduction

The term cryptography commonly refers to three related but independent functions: authentication, non-repudiation, and confidentiality. All three of these systems have their places in modern web application technology providing critical functionality for the security of the system. We'll investigate attacks against each of these three areas in this material.



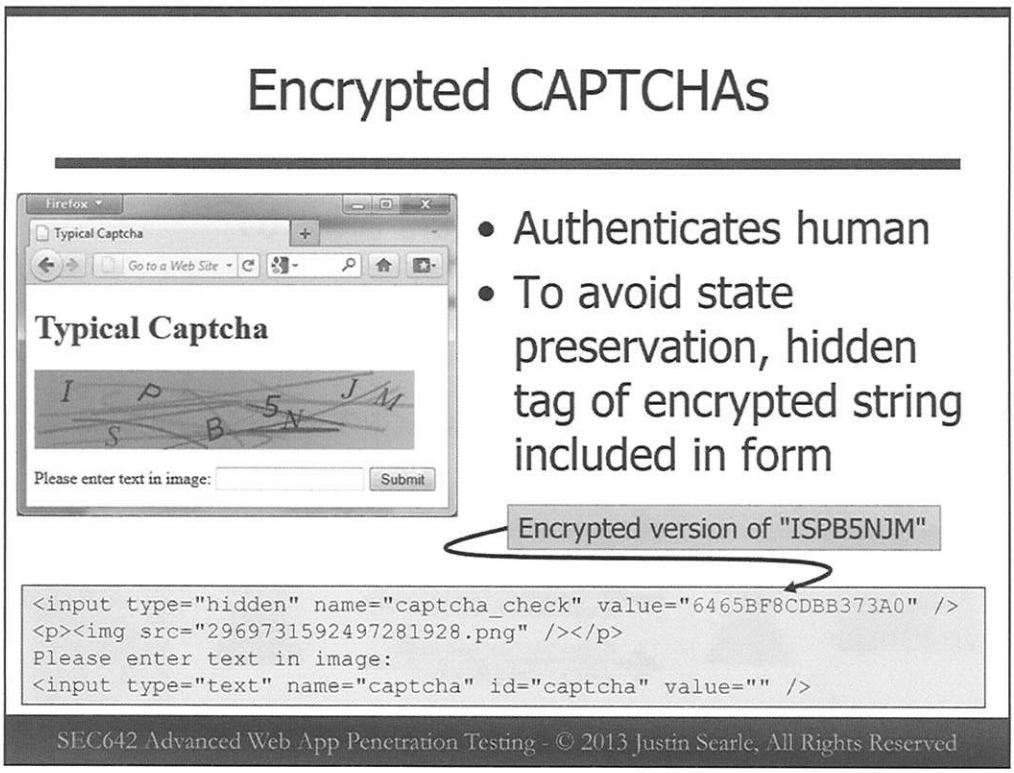
Authentication Controls

While not specifically inherent to cryptography, authentication systems commonly use a cryptographic function as part of the authentication mechanism. While legacy protocols such as SMTP, POP3, SNMP and FTP send plaintext credentials over the network to authenticate an entity which is vulnerable to network eavesdropping attacks, modern systems use two forms of cryptography to protect authentication credentials.

- **Transport Confidentiality:** Legacy protocols such as those previously mentioned also can be used in a more secure fashion by tunneling the weak authentication protocol inside a secure, encrypted channel such as SSL/TLS. This security mechanism is known as transport confidentiality and is widely used.
- **Secure Authentication:** More modern authentication protocols leverage a secure authentication exchange, commonly encrypting authentication credentials along with other unique data entities. This type of authentication had widely superseded weaker, legacy authentication protocols, while also optionally leverage transport confidentiality as well.

The illustration on this slide shows a simple example for using encryption in a secure authentication protocol:

1. The authenticator generates a random challenge value.
2. The authenticator sends the challenge and to the claimant.
3. The claimant AES-encrypts the challenge value 3 times using a key that is also known to the authenticator.
4. The claimant delivers the encrypted challenge to the authenticator. Note that the key used to encrypt the challenge is not sent over the network.
5. The authenticator similarly encrypts the challenge using the same key used by the claimant and compares the computed response to the observed response.
6. If the computed response and the observed response match, the authenticator sends an authentication success message to the claimant.

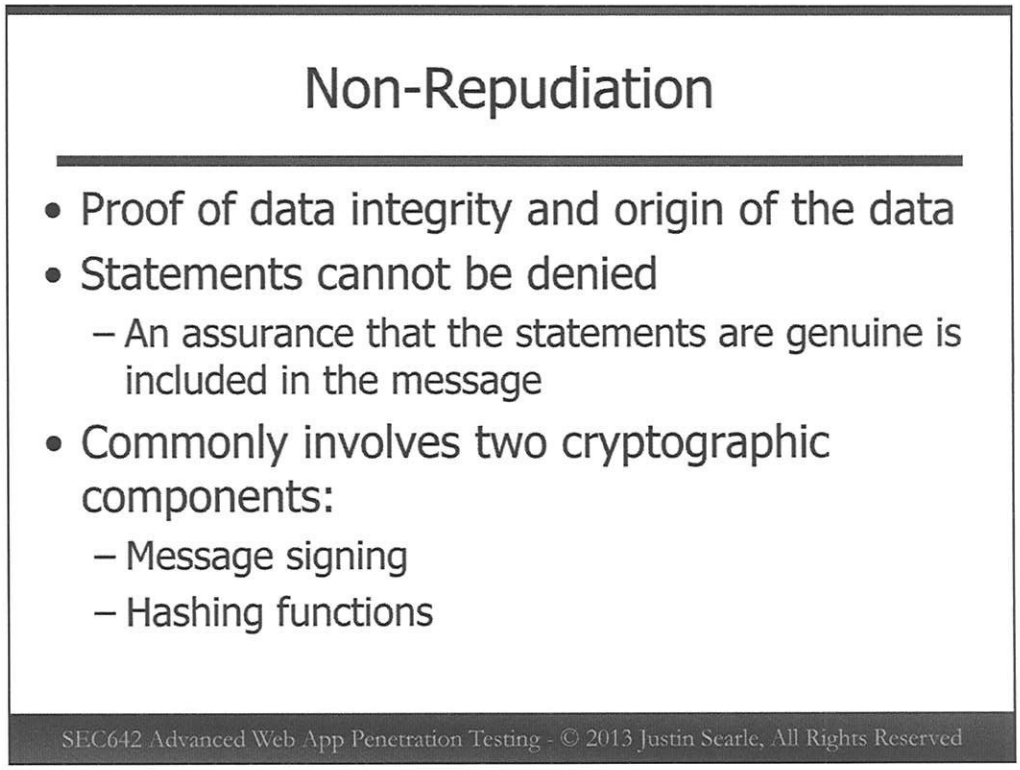


Encrypted CAPTCHAs

A CAPTCHA is a method to authenticate a person as opposed to a computer. The user is shown a distorted image such as the one on this slide, and asked to enter the text shown. Since it is difficult for a computer to interpret the distorted image, the site has a reasonable assurance that the person entering the content is indeed a person and not an automated system.

While several methods are available for the creation and use of CAPTCHAs, a common mechanism is to include an encrypted version of the plaintext string in the form through a hidden element. In the example on this slide, the HTML behind the CAPTCHA form includes a hidden element with the name "captcha_check" and a value of "6465BF8CDBB373A0". This value represents the encrypted form of the image string "IPSB5NJM". As both the encrypted version and the user-submitted plaintext version of the CAPTCHA are returned when the form is submitted, the developer does not need to maintain any state associated with the CAPTCHA challenge, simply decrypting the hidden element and comparing the plaintext string to the value entered.

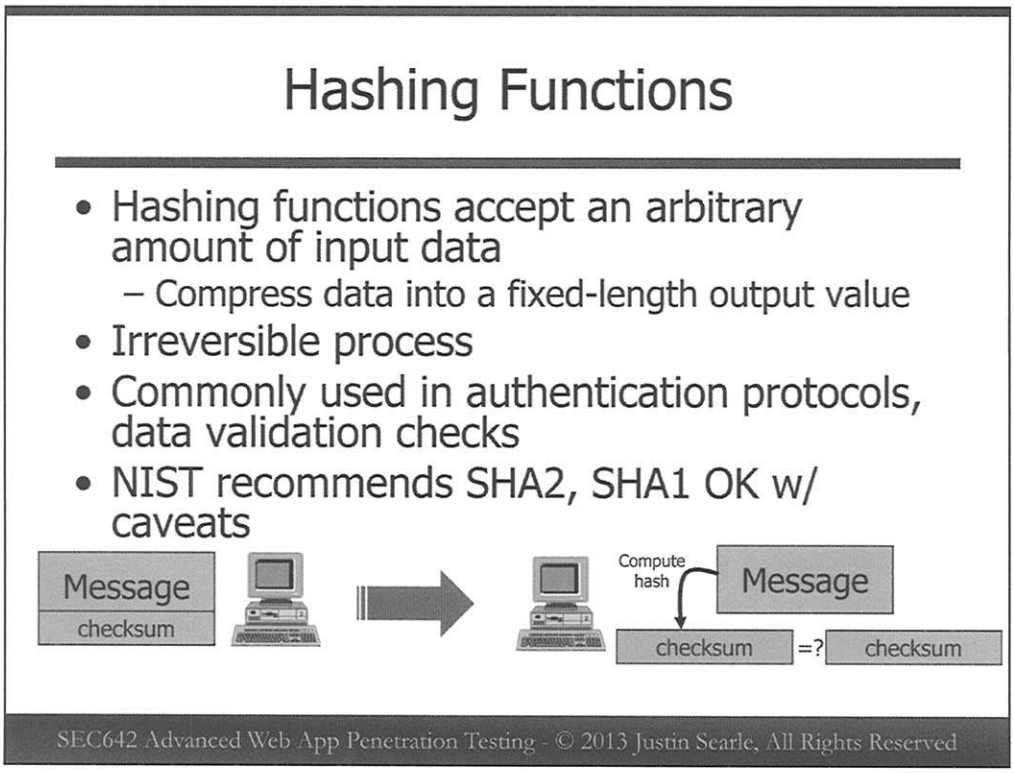
An example of this type of CAPTCHA challenge for the ASP.NET platform is available at http://haacked.com/archive/2006/10/02/Better_CAPTCHA_Through_Encryption.aspx.



Non-Repudiation

A second component commonly associated with cryptography systems is non-repudiation. Through non-repudiation, we obtain proof of data integrity and the origin of data, whereby the person originating the data cannot deny that they sent it.

In web application systems, non-repudiation provides a "proof" system where data cannot be modified in transit from the source to the destination. We commonly implement non-repudiation systems through two cryptographic components: message signing and hashing functions.



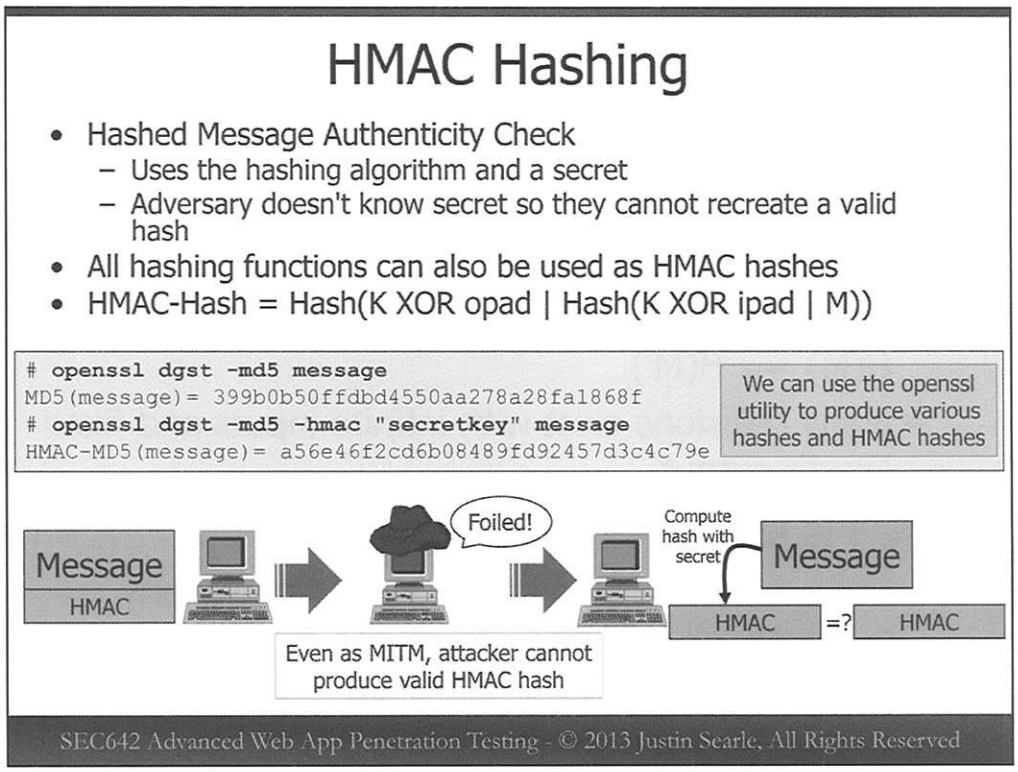
Hashing Functions

A hashing function accepts any amount of input data and compresses the data into a fixed-length output value that uniquely represents the input data. If one bit in the input data changes, then the output hash should have a dramatically different value (known as the avalanche effect). A hashing function is said to be non-invertible; that is, it is an irreversible process that, even with the hash value you cannot recover the original plaintext (notwithstanding brute-force guessing attacks).

In web application systems, a hashing function is a common component of authentication protocols and data validity checks. In the illustration on this slide, the transmitter (on the left) takes a message and computes a hash over the message which makes up the message checksum. Both the message and the checksum are sent to the recipient, who computes his own hash over the received message (not including the checksum). If the computed hash matches the checksum in the message, then the recipient believes the message is valid.

The National Institute of Standards and Technology (NIST) approves five hashing functions as secure including SHA1 (with some caveats and use constraints), and four hashing functions from the SHA2 family namely SHA-224, SHA-256, SHA-384, and SHA-512. Additional information on NIST validation of hashing functions is available on the NIST website at http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

Other hashing functions include MD2, MD4, MD5, ripemd160, MDC2 and the SHA function.



HMAC Hashing

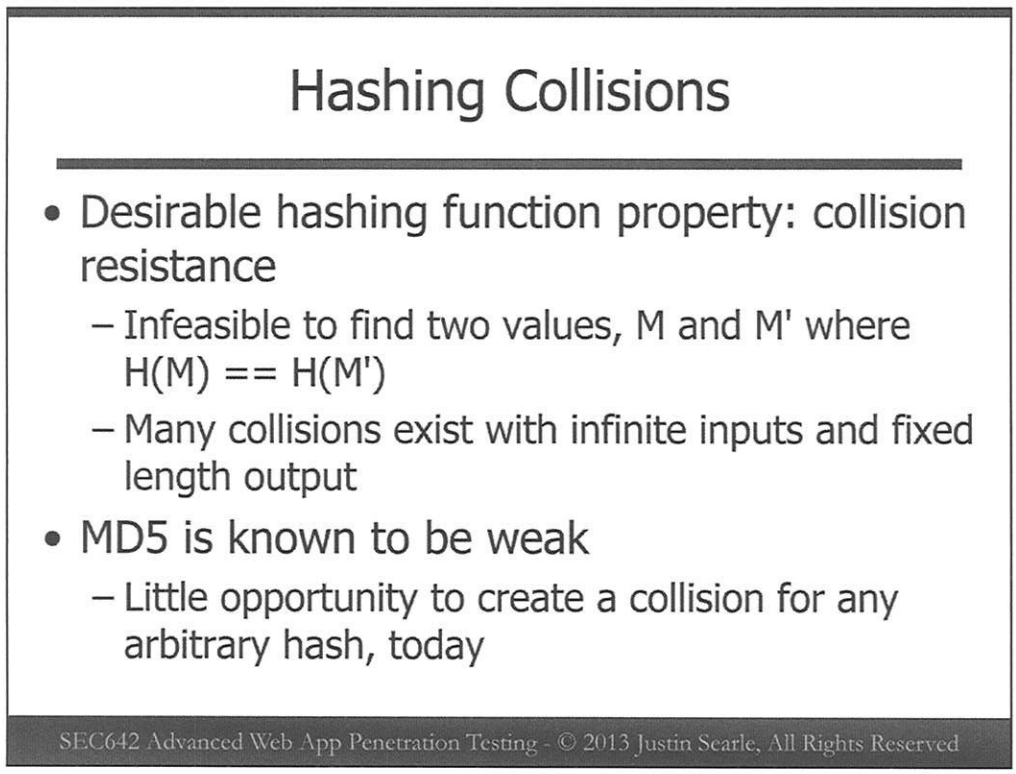
There is a significant problem in the use of hashes for valid authentication as shown in the previous slide. When the transmitter includes the hash along with the original content, a Man-in-the-Middle (MitM) attacker can intercept the message and hash prior to delivery, change the message and simply replace the original hash with one that is computed over the changed content. Upon receipt, the victim does not know that the message has been modified, only that the sent hash is correct.

To address this limitation, the IETF developed a method of hashing known as a Hashed Message Authenticity Check (HMAC) hash. In a HMAC hash, the transmitter creates a hash with the assistance of a secret value known to the transmitter and the recipient. In order to recompute the hash, the attacker must also know the secret value, preventing them from creating valid hashes for intercepted data.

The HMAC function is independent of the hashing method itself, using the hashing function as a naming suffix (e.g. HMAC-MD5, HMAC-SHA512, etc.) Any hashing function can also be used as an HMAC hash.

The HMAC hash is computed using the hashing function twice, as shown on this slide. Initially, the secret "K" is XOR'd with a fixed value known as the ipad (repeating 0x36 bytes) before being concatenated with the message M. This value is then hashed using the named hashing function (e.g. MD5, SHA512). After computing the inner hash, the secret "K" is XOR'd with the fixed value known as the opad (repeating 0x5c bytes) before being concatenated with the inner hash. The resulting value is then hashed using the named hashing function a second time to produce the HMAC hash.

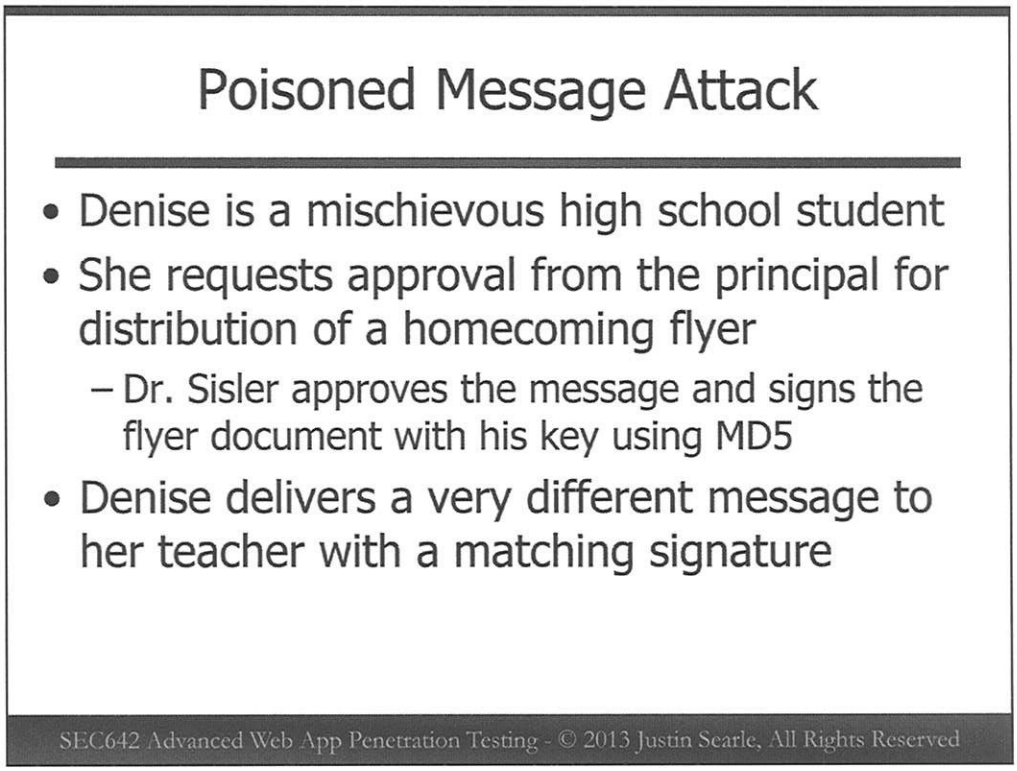
While most Unix operating systems include tools to compute SHA1 or MD5 hashes over arbitrary content (sha1sum, md5sum), these tools do not accommodate the computation of HMAC hashes. To compute a HMAC hash, we can use the openssl command-line utility as shown on this slide, specifying our secret value on the command-line and the filename of the content to hash ("message" in this example).



Hashing Collisions

For any cryptographic hashing function, a desirable property is that the function be resistant to collisions. Since a hashing function takes any amount of input data and produces a fixed-length hash as the output, there will always be inevitable collisions, but these collisions should be infeasible to discover through computational analysis. Secure hashing functions prevent an attacker from identifying two values, M and M' ("M prime") where the hash of M is equal to the hash of M'.

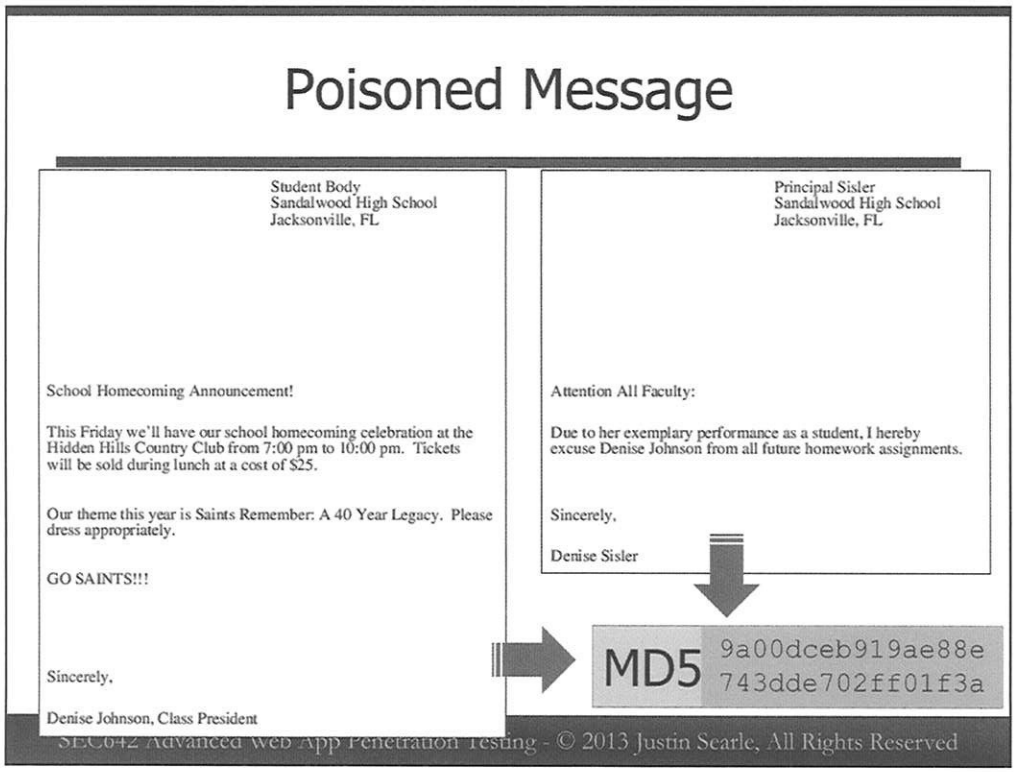
Several hashing functions fail this expectation today with MD5 being the most commonly used. It is important to note however that even though MD5 is weak, there is little opportunity for an attacker to generate a matching MD5 hash for an arbitrary file. However, if an attacker can influence the initial file content (M), it is possible for them to create a second file (M') that will produce an identical MD5 hash.



Poisoned Message Attack

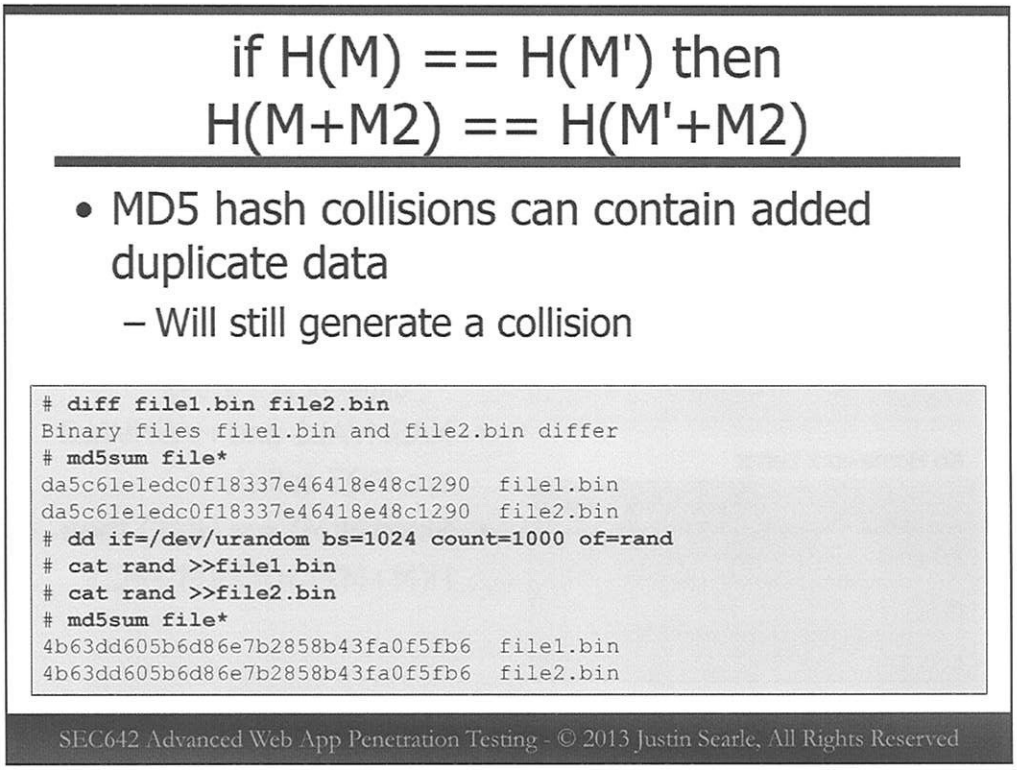
The ability to produce a hash collision has led to poisoned message attacks. For example, consider the case of Denise, a mischievous high school student. Denise gets approval from her principal, Dr. Sisler, for the distribution of a flyer advertising the school's homecoming dance event. Dr. Sisler reads the message and approves, signing the message with his key using MD5.

However, Denise's homecoming flyer was created specifically such that it would produce a MD5 collision that she could control to leverage the signature from Dr. Sisler, but have very different message content.



Poisoned Message

Denise's poisoned message signed by the principal, Dr. Sisler, is shown here in two forms. On the left is the message content observed and approved by Dr. Sisler advertising the homecoming event. The message on the right releases Denise of any homework responsibilities. Both messages produce identical MD5 hashes.

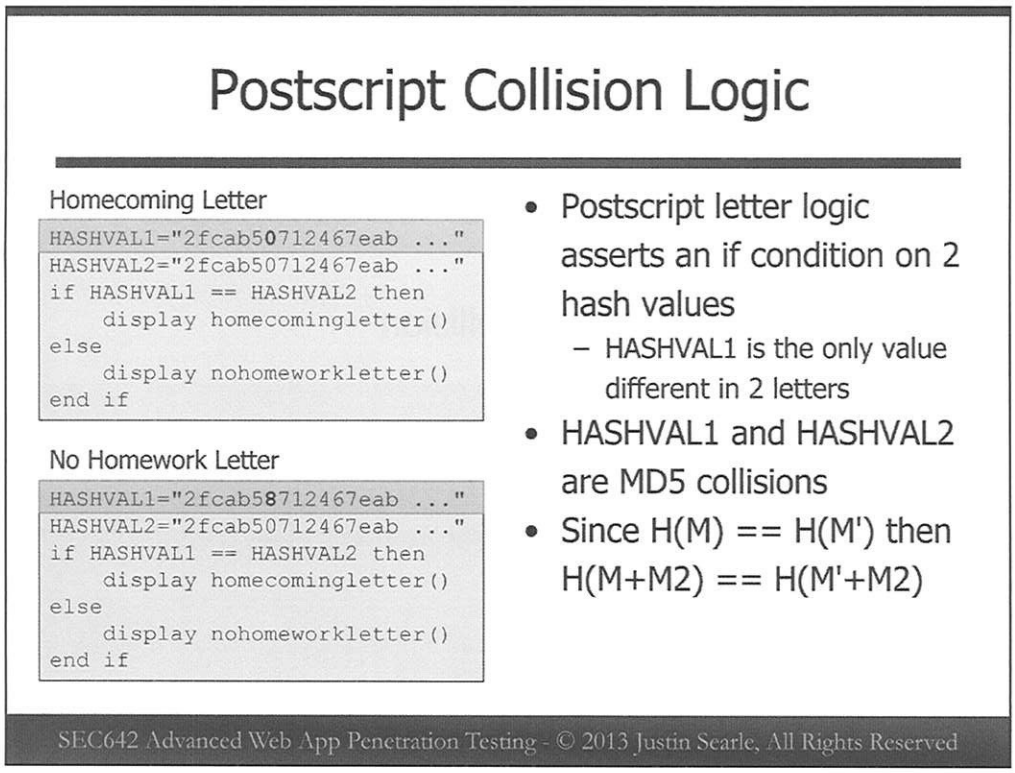


Hash Collision Logic

A common weakness in hashing functions is that, when two different messages produce a hash collision, they can be modified by adding identical data and still produce hash collisions. That is if a Hash of M is equal to a Hash of M', then a hash of M with the addition of message 2 (M2) is equal to a hash of M' with the same additional message (M2).

This is the case for MD5, and the trick behind Denise's letter. In the example on this slide, we see the two input files, file1.bin and file2.bin. While these files are different (as identified by the diff utility), computing the MD5 hash on these two files shows that they have the same hash value.

Next we generate a second file ("rand", representing M2), appending the content to both original input files. Computing the hashes for both files again we see that while the MD5 hash is different, is it still consistent between the two files.



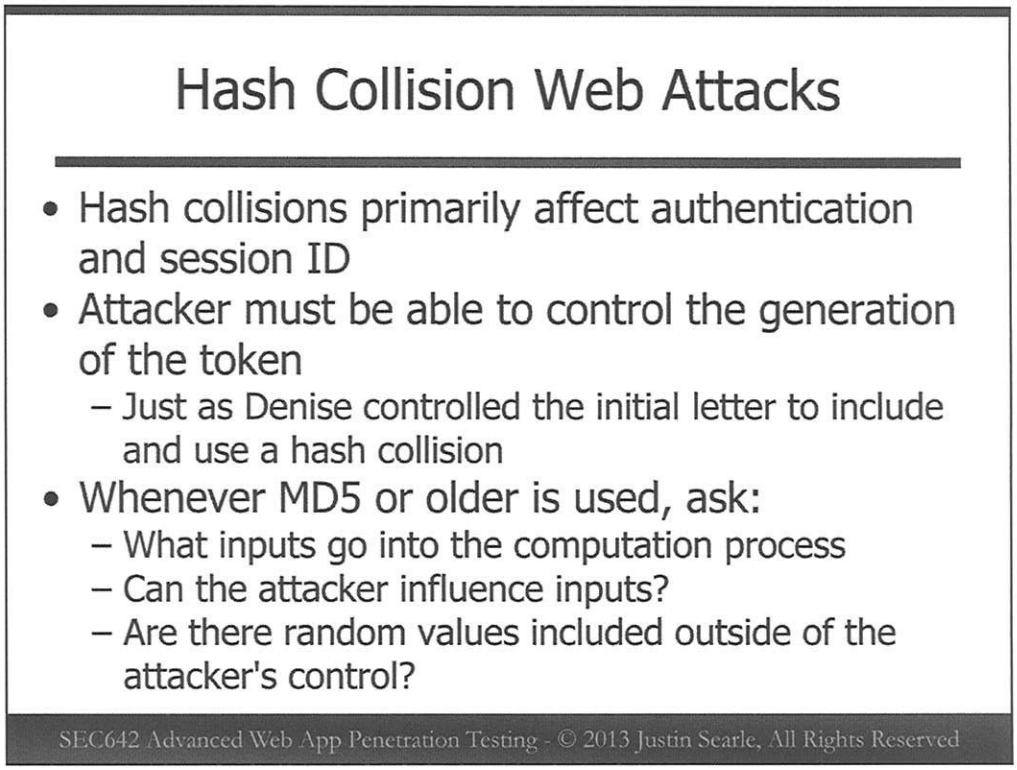
Postscript Collision Logic

In order to produce her two letters with the same hash, Denise created a PostScript file with two values that will produce a hash collision. Since PostScript is essentially a programming language, she used logic within the document to create two variables, "HASHVAL1" and "HASHVAL2". In the first document, HASHVAL1 and HASHVAL2 are identical; the comparison operator then displays the homecoming letter content when rendered.

In the second example however, the HASHVAL1 is different by HASHVAL2 by only one bit. However, this is sufficient for comparison operator to display the no homework letter instead when rendered.

Note that these two documents are identical with the exception of the HASHVAL1 value. Since HASHVAL1 and HASHVAL2 (in the 2nd letter) create a hash collision, and the remainder of the document is identical (representing M2), then the MD5 hashes of both documents are identical.

A similar technique could be applied to PDF documents, or possibly Microsoft Word documents as well, as these document formats all accommodate the author's ability to programmatically select the content to display.

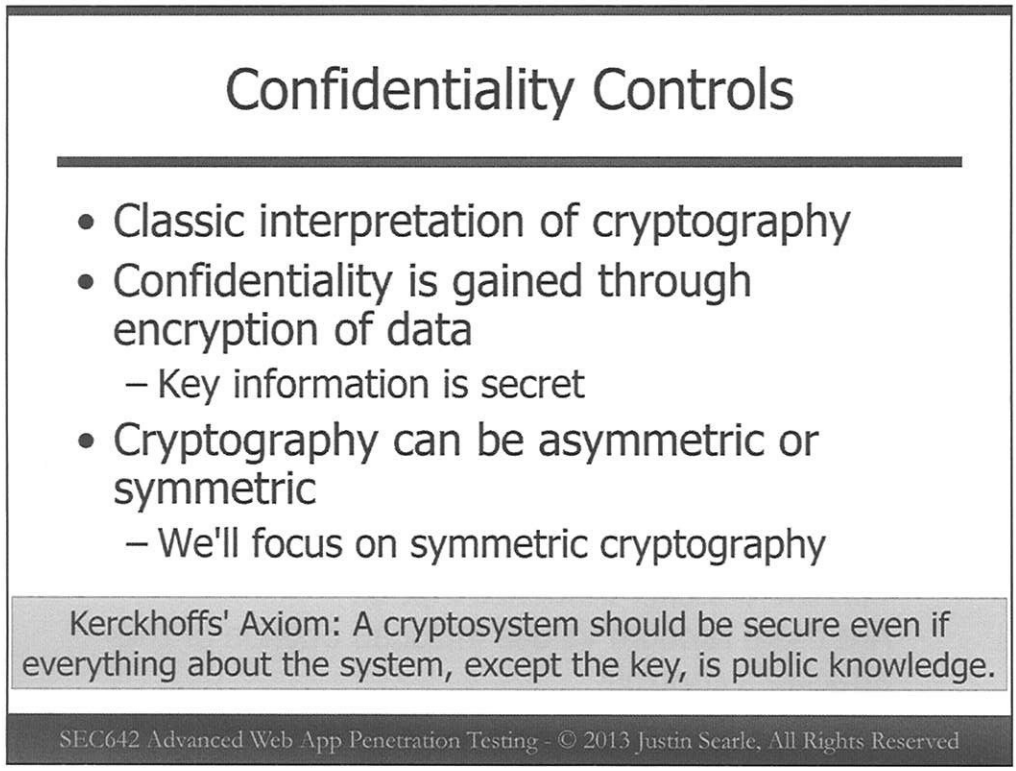


Hash Collision Web Attacks

To date we have not seen the widespread application of hash collision attacks against web applications, though it could be applicable to interfaces affecting the authentication of a session or session ID values.

Remember that in order to create a hash collision, the attacker must be able to control the content that is used for the generation of the token value that is hashed. This is unlikely in scenarios where the web server generates the data to be hashed, but is possible in other situations where the attacker can influence the data that is hashed (such as entries in a database table through SQL injection).

In any situation where MD5 or older hashing functions such as MD4 or MD2 are used, review the inputs that go into the computation process to ensure that they cannot be controlled by an attacker.

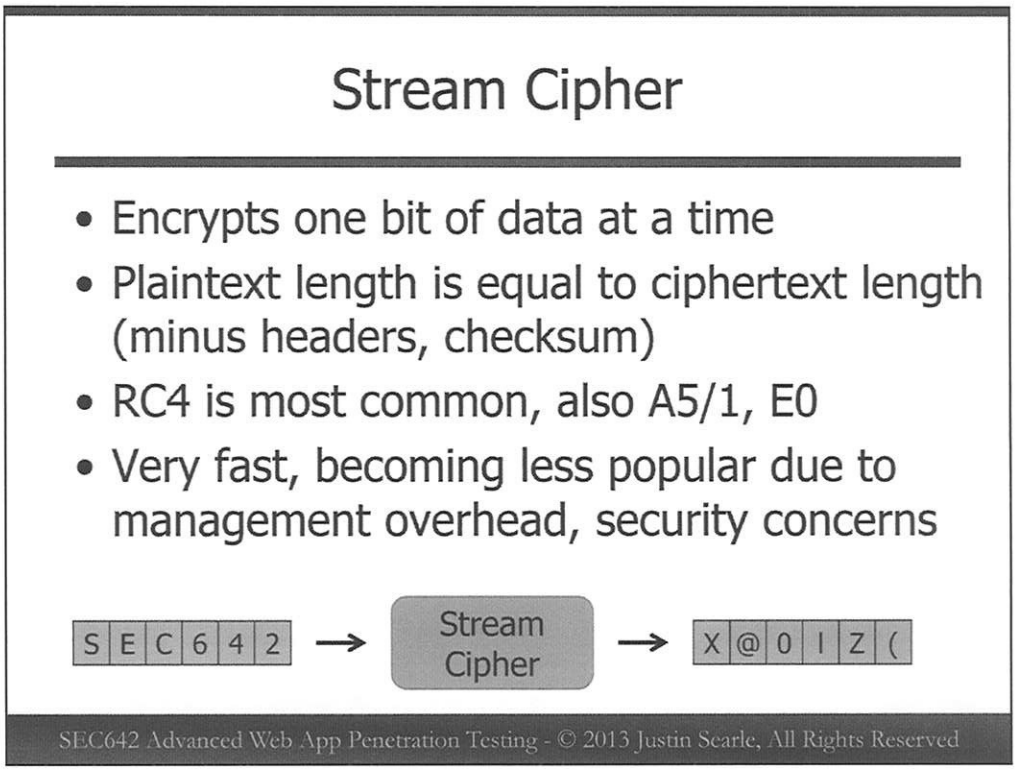


Confidentiality Controls

The classic interpretation of cryptography is the ability to provide confidentiality controls to data through encryption and decryption with the use of a secret key. Encryption functions can be asymmetric (where two keys are used; one for encryption and a second key for decryption) or symmetric (where a single key is used for both encryption and decryption). In this course we'll focus on symmetric cryptography systems since it is the most applicable to web application systems.

A notable point in our discussion of confidentiality controls is Kerckhoffs' Axiom named after Dutch linguist and cryptographer Auguste Kerckhoffs, stating that cryptosystems should be secure even if everything about the system, except the key, is public knowledge [1]. No obscurity should be relied upon for the security of the cryptosystem, other than the privacy of the key itself.

[1] Auguste Kerckhoffs, "La cryptographie militaire" Journal des sciences militaires, vol. IX, pp. 5–83, January 1883, pp. 161–191, February 1883.

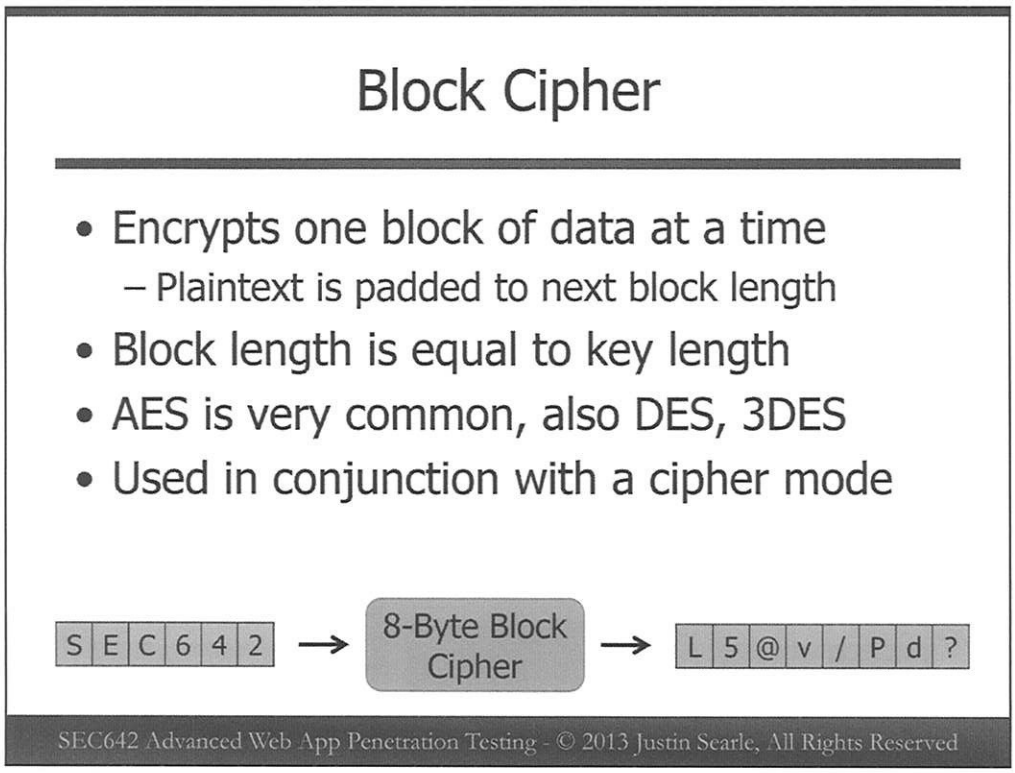


Stream Cipher

Included amongst symmetric encryption options are **stream ciphers**. In a stream cipher, the input data is encrypted one bit at a time without adding additional encrypted content for each bit of input plaintext. This gives a stream cipher the property of identical length of plaintext and ciphertext data, possibly excluding protocol header information or checksums.

Of stream ciphers, RC4 is the most commonly used, popularly for SSL/TLS encryption. The A5/1 stream cipher is also very popular used for GSM phone calls and SMS messages, while the E0 stream cipher is used to protect Bluetooth transmissions.

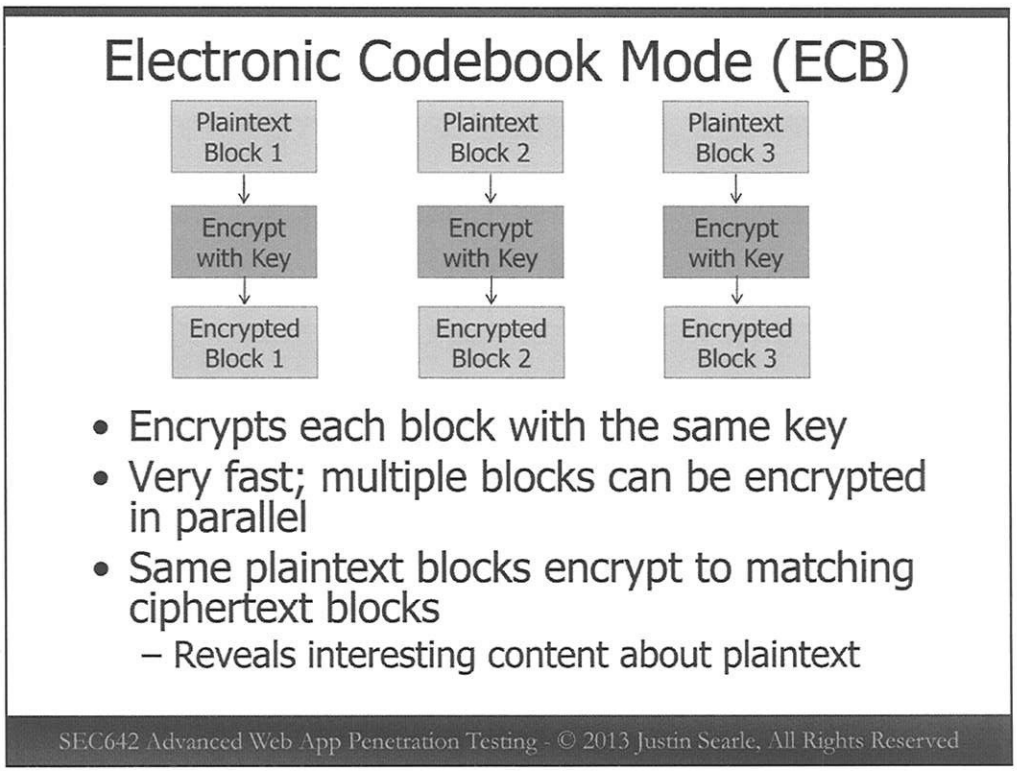
A stream cipher considered to be very fast as an encryption function, but is becoming less popular due largely to added protocol requirements for use and security concerns.



Block Cipher

In contrast to a stream cipher, a block cipher encrypts one block of data at a time, typically 8 or 16 bytes, matching the length of the key. A block cipher cannot encrypt content that is shorter than the block length, so it uses a padding function to increase the final block of plaintext to an even block length prior to encryption.

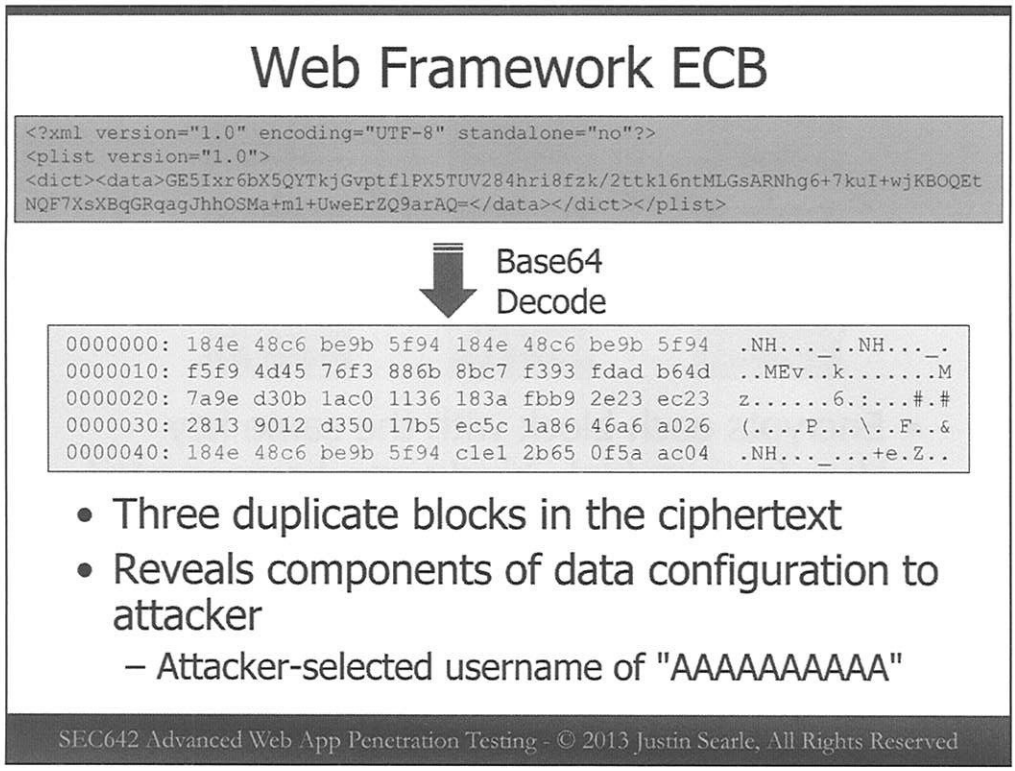
AES is a very common block cipher, as are DES, 3DES, Blowfish and others. When a block cipher is used to encrypt data, it is done so in conjunction with a block cipher mode that influences how the data is encrypted.



Electronic Codebook Mode (ECB)

The simplest and most straightforward block cipher mode is Electronic Codebook Mode (ECB). In ECB mode, each block is encrypted independently of other blocks, using the same key. This operation is very fast since multi-processing systems can encrypt the blocks in parallel.

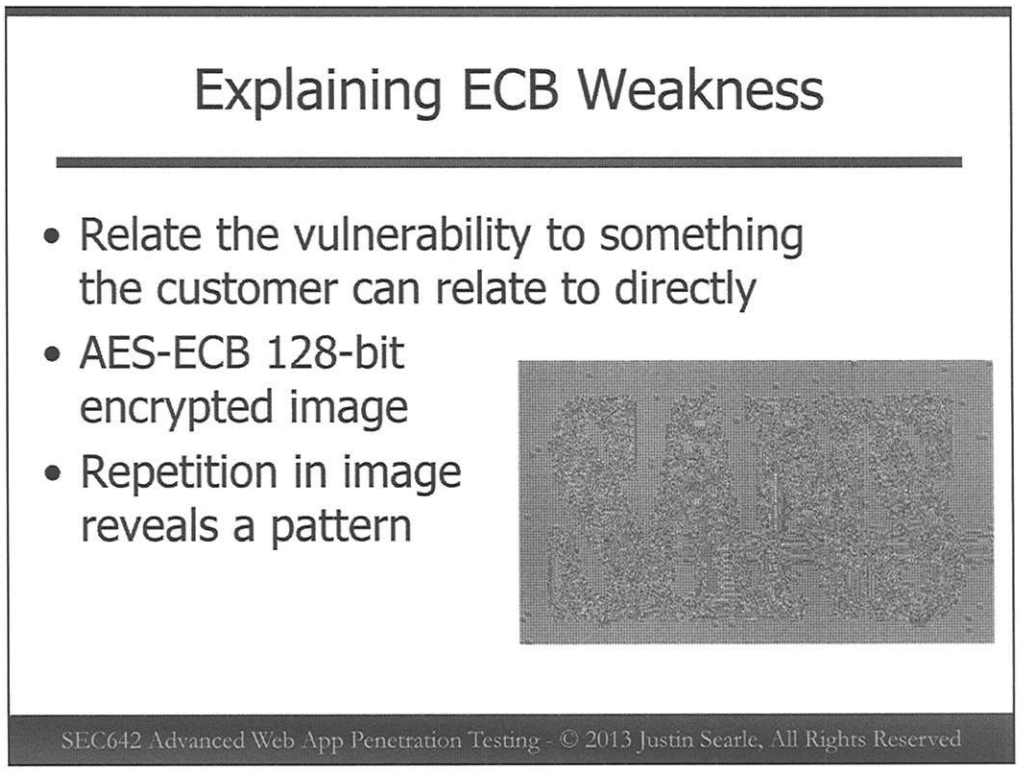
ECB is known to be vulnerable to several attacks however, including the ability for an attacker to identify duplicate ciphertext blocks as indicating duplicate plaintext blocks as input to the cipher.



Web Framework ECB

The use of ECB mode can be identified in web frameworks through the repetition of blocks in encrypted data. In the example on this slide, a web framework observed by the author uses an XML data structure to exchange data between a supporting server and a handheld mobile device. The XML data content includes an element "data" which is a base64-encoded string.

After decoding the string, we can examine the contents of the data in a hexdump format. Quickly we can see that three of the 10 blocks are duplicate values. This is a common effect of ECB mode, where duplicate plaintext blocks generate duplicate ciphertext output. In this scenario, a username of "AAAAAAAAAA" was used as input in an attempt to identify the encryption mechanism in use, possibly indicating that the username is partially contained in the three duplicate blocks.

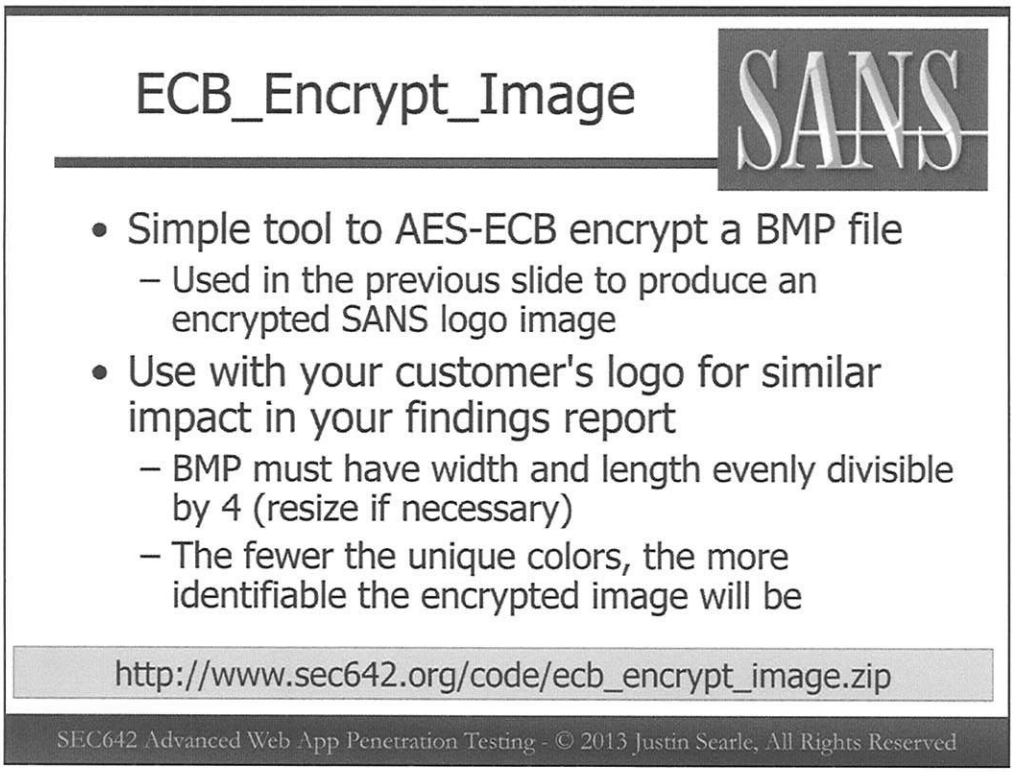


Explaining ECB Weakness

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. Similarly, an attacker with access to large quantities of ECB encrypted data can infer information about the plaintext content of the date.

Through the use of this visualization tool we can help our customers 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.

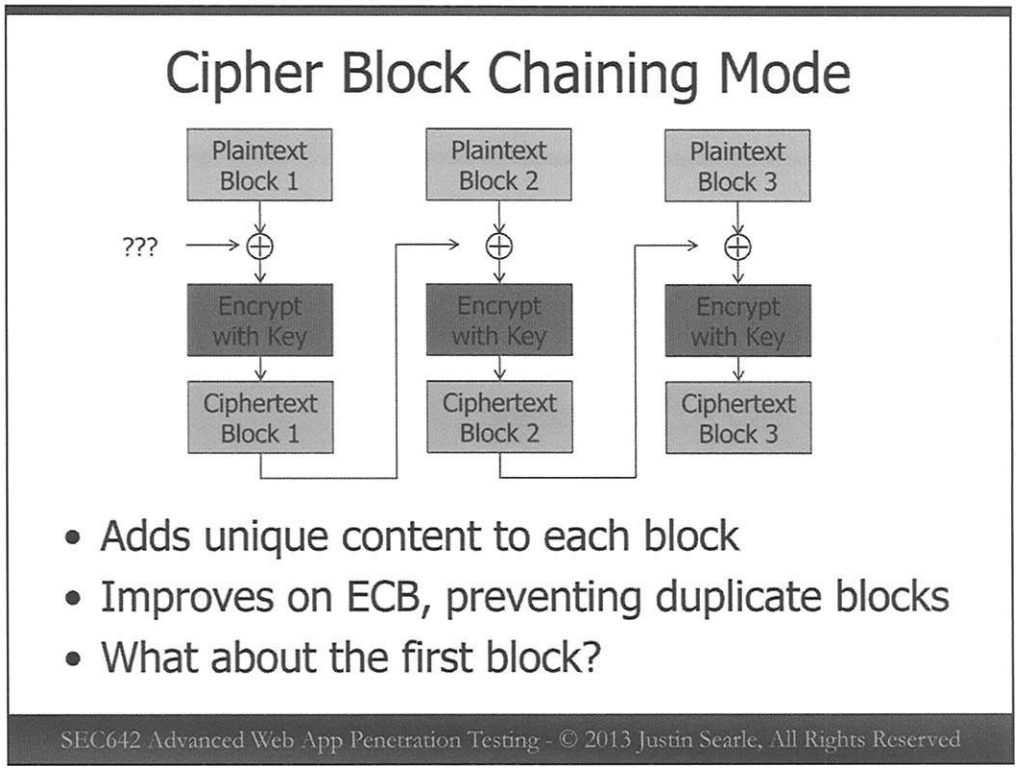


ECB_Encrypt_Image

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 fewer the total color count, the less uniqueness there is in the file, producing more revealing encrypted bitmaps.

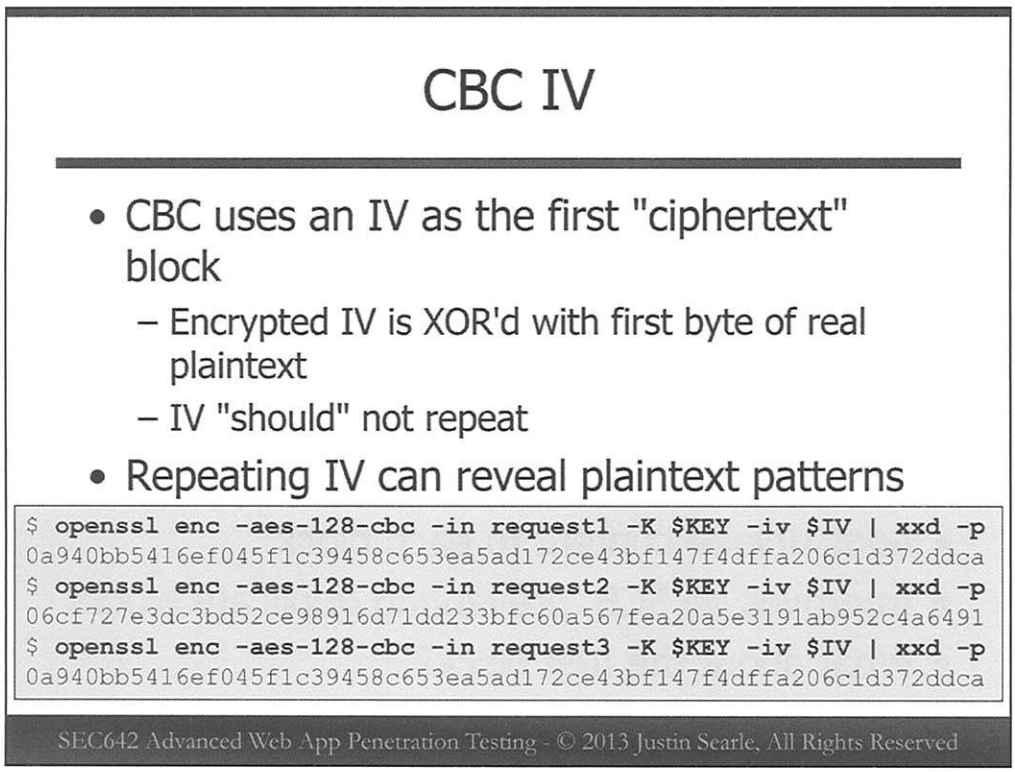


Cipher Block Chaining Mode

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 XOR'd 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 unique input to each encryption operation. This improves on the ECB mode by preventing the presence of duplicate blocks.

One concern however is how the first block of plaintext is encrypted. Since each block of plaintext is XOR'd with the prior ciphertext block, we have a problem with the first block since no prior encrypted content is available. To solve this problem, we use a special value known as the initialization vector (IV) the length of the block size as the XOR input with the first block.



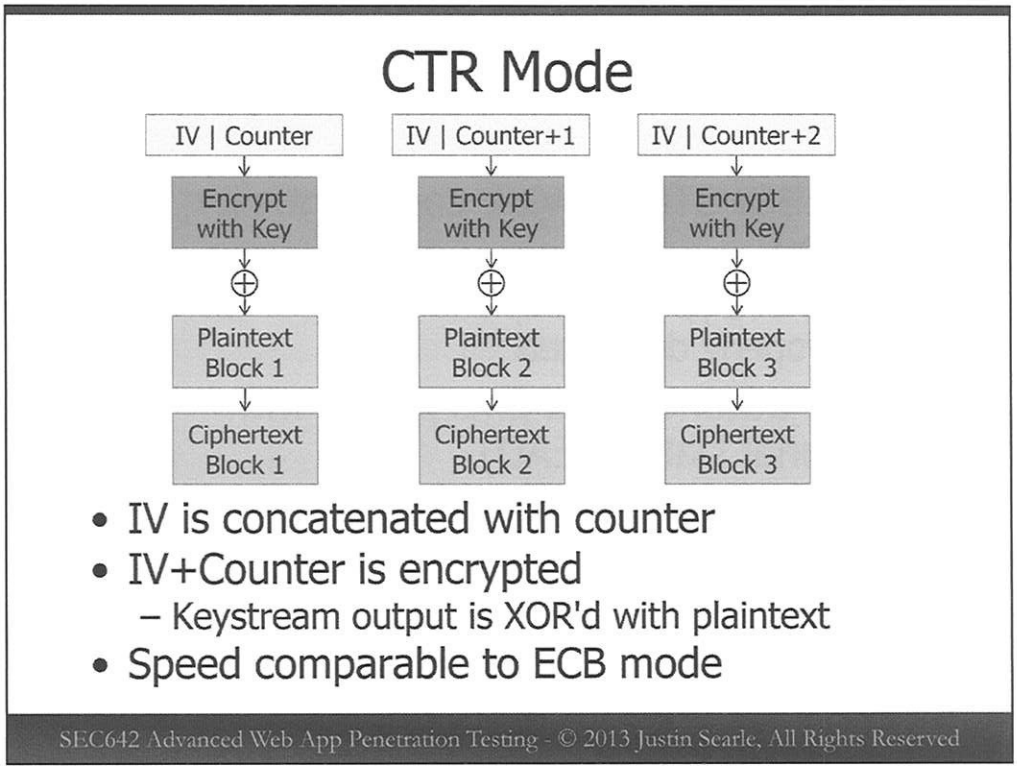
CBC IV

CBC requires the use of an IV to encrypt the first plaintext block. Each successive plaintext block is XOR'd 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 must not 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 requests 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 requests, dumping the output in hex with the xxd utility. The output of these three encrypted requests repeats for packet 1 and packet 3, revealing to the attacker that the plaintext content of these two requests is the same.

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



CTR Mode

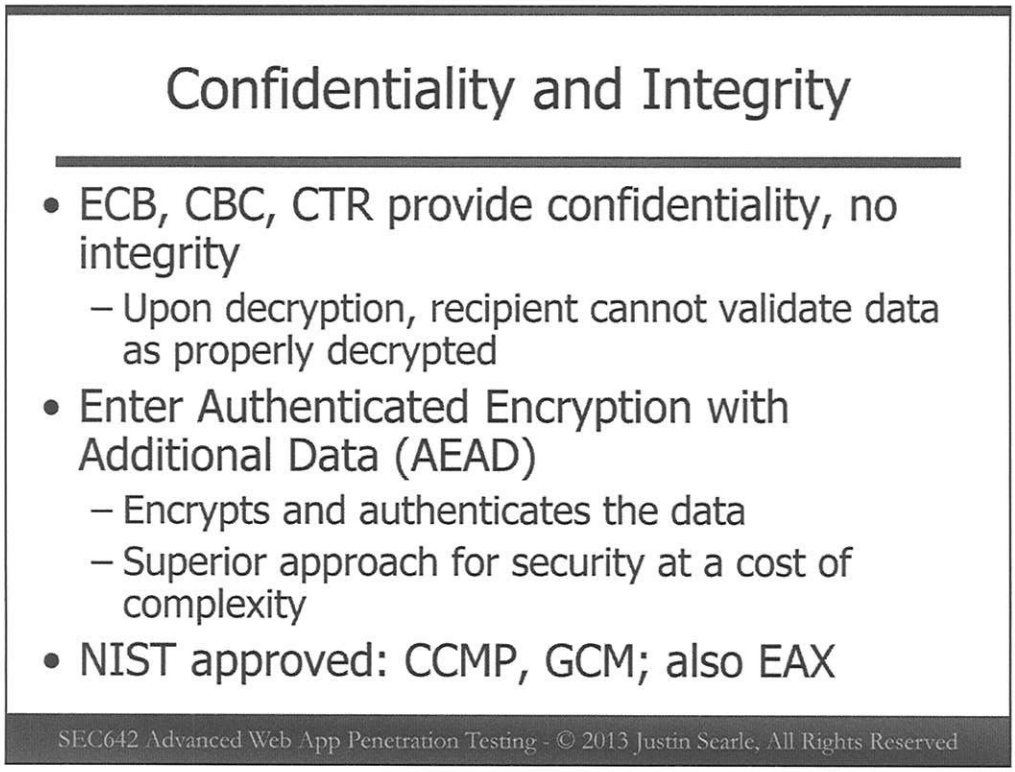
In counter mode, we continue to use an IV, but we do not feed the output of the prior ciphertext to the next encryption block. Instead, the IV is concatenated with a counter value that represents the input for the encryption algorithm. In this mode, the IV is not the length of the block but is instead 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 XOR'd 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 XOR'ing 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 section we'll look at techniques to exploit stream ciphers and block ciphers in stream cipher mode (including CTR mode) when an IV is reused.

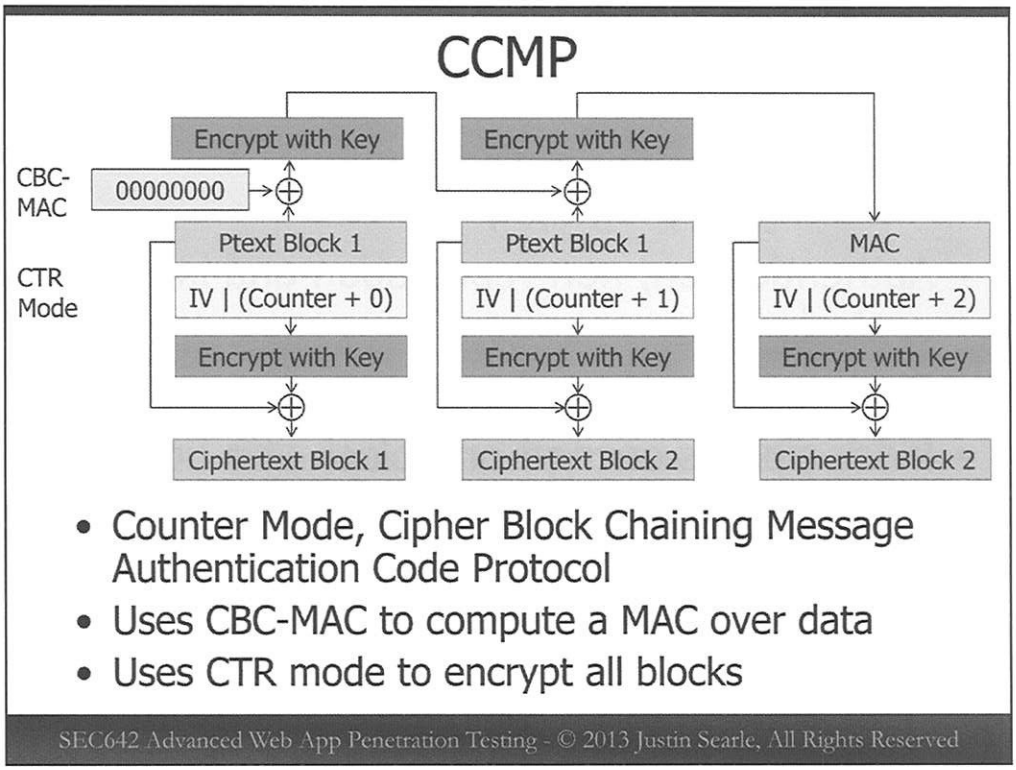


Confidentiality and Integrity

So far we've examined the operation of the ECB, CBC and CTR modes used with block ciphers. These three modes provide varying levels of confidentiality, but do not provide any guarantee of integrity. That is, while the recipient and transmitter have some assurance that the data remains confidential in transit, there is no guarantee that the data has not been modified in transit and will decrypt to the intended plaintext value.

To achieve both confidentiality and integrity protection, we leverage Authenticated Encryption with Additional Data (AEAD) encryption modes. AEAD provides both encryption for confidentiality of the data, as well as a data integrity check that provides a cryptographically secure validation mechanism to identify data that has been modified in transit. AEAD provides a superior level of security over standard block cipher modes, but does so at the cost of complexity for the system.

The National Institute of Standards and Technology (NIST) approves two AEAD modes for use: Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP) and Galois/Counter Mode (GCM). A third AEAD mode known as EAX (not an acronym) was also submitted for NIST approval in 2003 and is commonly used for embedded systems for the protection of Supervisory Control And Data Acquisition (SCADA) and Advanced Metering Infrastructure (AMI) related data systems.

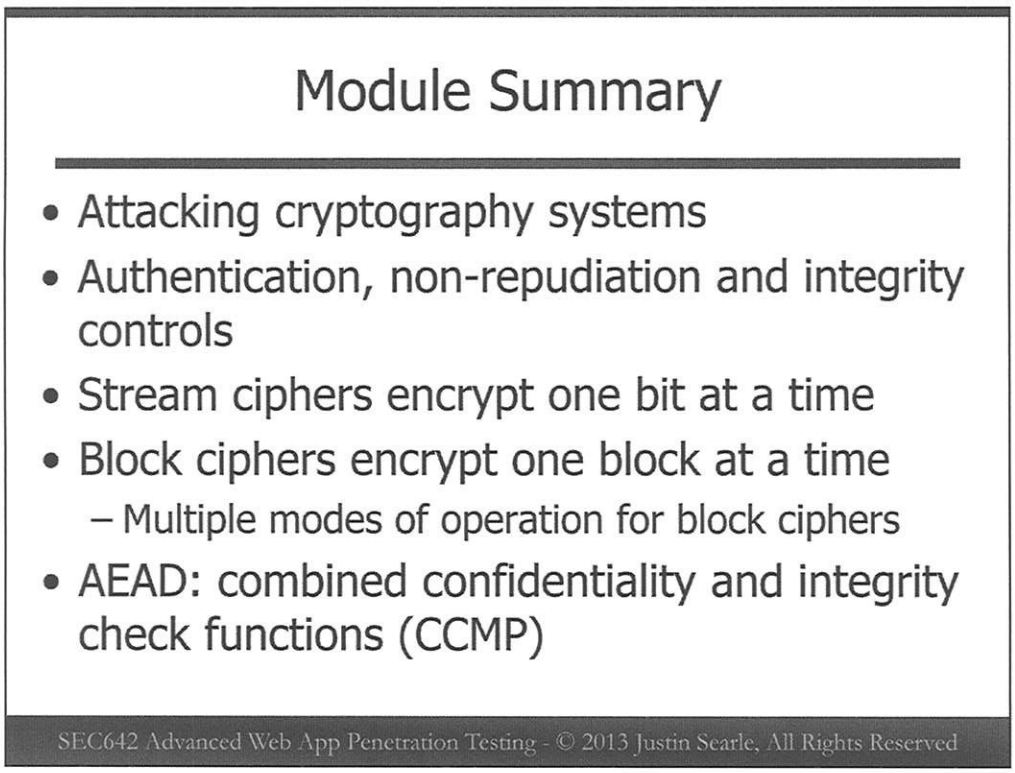


Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP)

CCMP is the most popular AEAD method used today and can be found in some web applications. CCMP uses CTR mode encryption to provide confidentiality for plaintext data where the IV is concatenated with a counter value that is incremented for each block. The combined counter and IV is encrypted and XOR'd with the plaintext to generate a ciphertext value.

In addition, the plaintext block is encrypted using a function known as CBC-MAC. CBC-MAC is essentially CBC encryption with two variations: the IV is consistently all 0's each time the CBC-MAC is computed, and none of the ciphertext blocks are retained in the encryption process except for the last block. In the example on this slide, the plaintext block is XOR'd with the initial static IV of all 0's before being encrypted with the key. This encrypted value is then used as the XOR input for the next plaintext block and is then dropped as it is no longer needed for the CBC-MAC computation. This process continues until the last block of data has been encrypted. The encrypted output of the last block is then appended to the plaintext value and encrypted using CTR mode.

Upon reception, the data is decrypted using the standard CTR mode of operation. The final block of the plaintext represents the CBC-MAC of the plaintext where the recipient computes the CBC-MAC over the received data and compares the computed CBC-MAC to the observed CBC-MAC in the last block of the plaintext. If the two values match, then the recipient knows the ciphertext was not modified in transit.

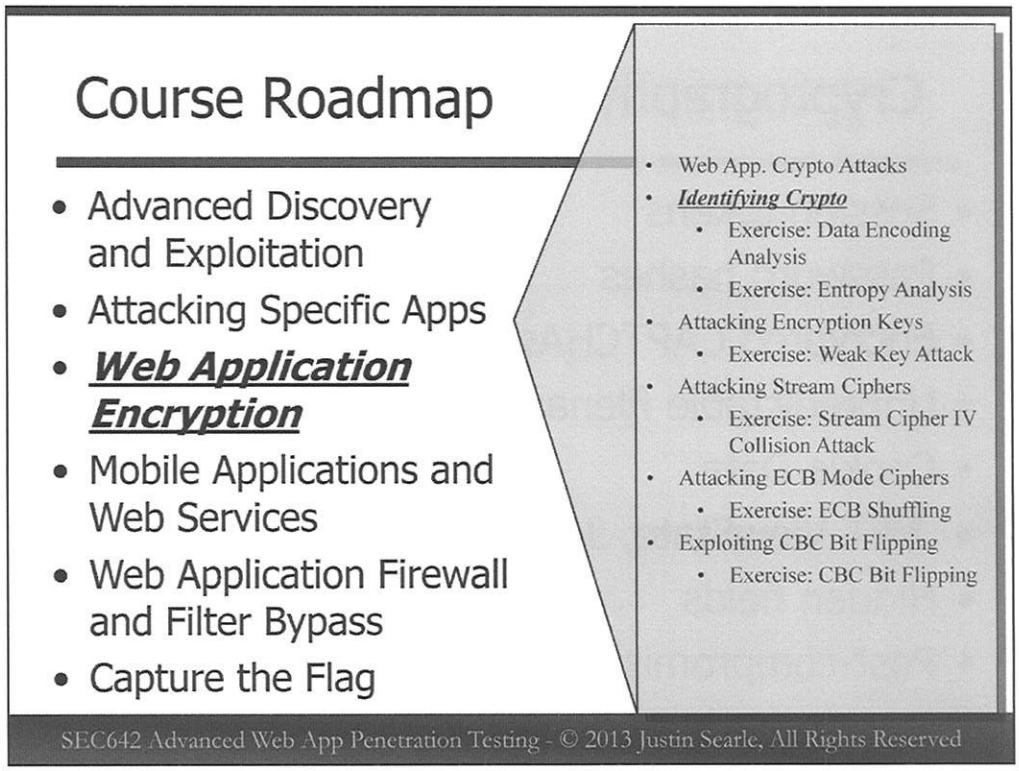


Module Summary

In this module we examine the concepts behind attacking cryptography systems by reviewing the fundamental concepts necessary for any penetration tester . At a fundamental level, cryptography systems include authentication, non-repudiation and integrity controls which can leverage encryption and decryption functions as well as hashing functions.

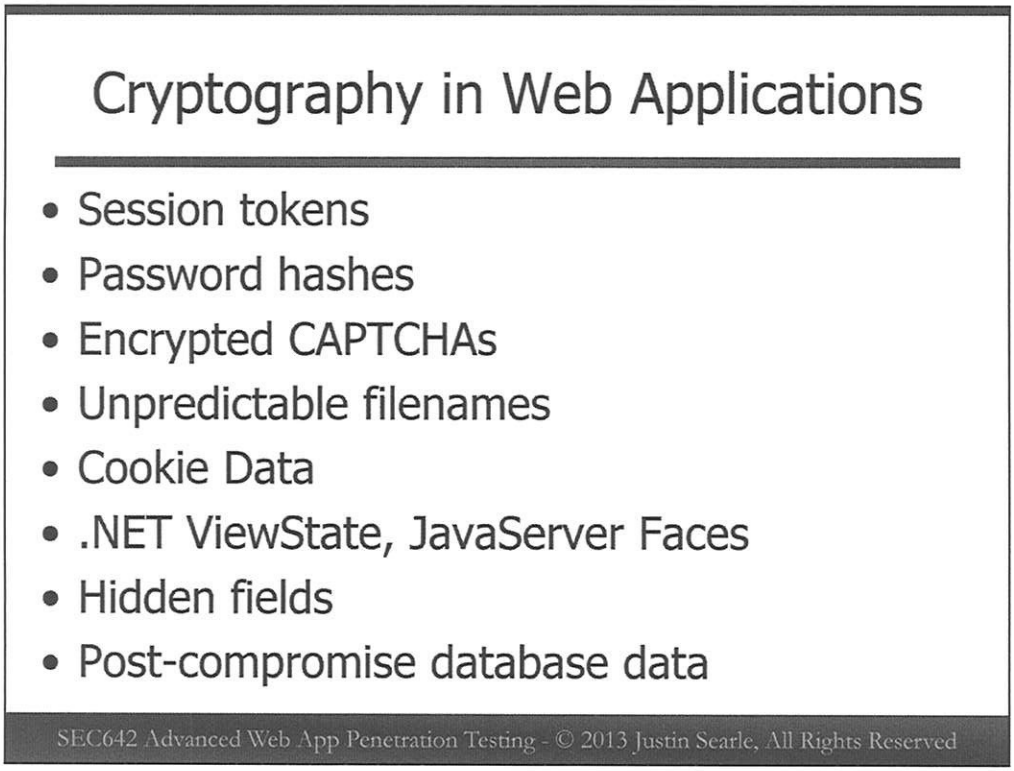
We examined two major varieties of encryption protocols including stream ciphers, which encrypt one bit of data at a time and block ciphers which encrypt one block at a time. Block ciphers additionally have a defined mode of operation, such as Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter (CTR) mode which defines how the cipher is used to encrypt the plaintext.

While a stream cipher or a block cipher can provide data confidentiality, it does not also guarantee data integrity at decryption. In order to achieve a cryptographically secure integrity mechanism as a component of the encryption protocol we use an Authenticated Encryption And Data (AEAD) function, such as Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP).



Course Roadmap

Next we'll look at how to identify and characterize the use of cryptography in web applications.



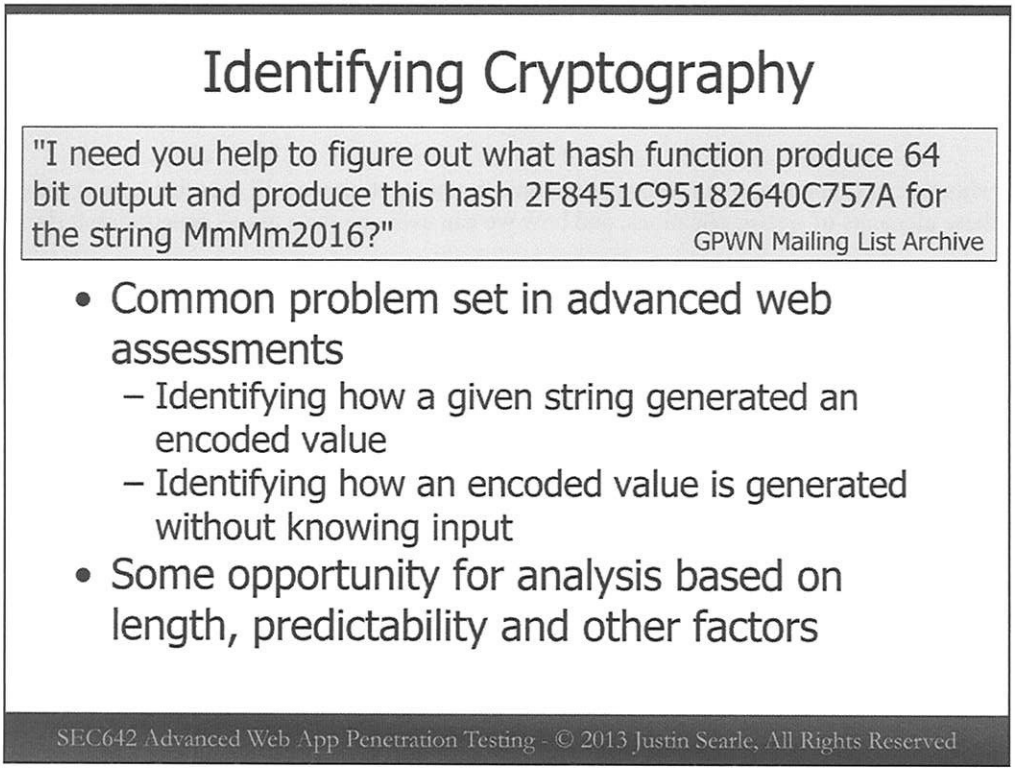
Cryptography in Web Applications

Having covered several of the fundamental concepts behind modern cryptosystems, we can look at how these system apply to web applications. Modern web applications and frameworks leverage some soft of cryptography almost universally in order to protect the confidentiality and integrity of system functions:

- Session Tokens: Since HTTP is a stateless protocol, a session management mechanism is necessary to keep track of the state of a connection. Session tokens are used in common programming frameworks such as PHP (in the PHPSESSID header) and are generated using common cryptographic hashing functions to represent but protect sensitive input values.
- Password Hashes: The storage of passwords in a hash format allows the system to leverage them for authentication while protecting their confidentiality. Password hash content is commonly found in custom web applications where authentication is used, and in many database systems supporting the web application.
- Encrypted CAPTCHAs: In order to avoid maintaining state across requests, some CAPTCHA forms will encrypt the ASCII version of a CAPTCHA and include it in the form response for the recipient to decrypt and validate.
- Unpredictable Filenames: Many web systems will use some sort of cryptography to protect against unauthorized access of hosted files in the selection of filenames or directory names.
- Cookie Data: Cookies are widely used in web applications, especially with their ability to store arbitrary content including encrypted data.
- .NET ViewState, JavaServer Faces: Web application frameworks such as ASP.NET and JavaServer Faces encrypt session information sent between the client and the server for the protection of sensitive content such as session privileges and authentication credentials.

- Hidden Fields: Hidden HTML fields are widely used by both web application frameworks (including ASP.NET) and custom applications for the storage of data including confidential fields.
- Post-Compromise Database Data: After exploiting a SQL injection vulnerability, a web application pen tester will commonly encounter encrypted database content in rows, columns or entire tables.

There is no shortage of cryptography in web applications. In this section we will examine how cryptography is used in all these elements of web applications, and how we can evaluate their use as penetration testers.

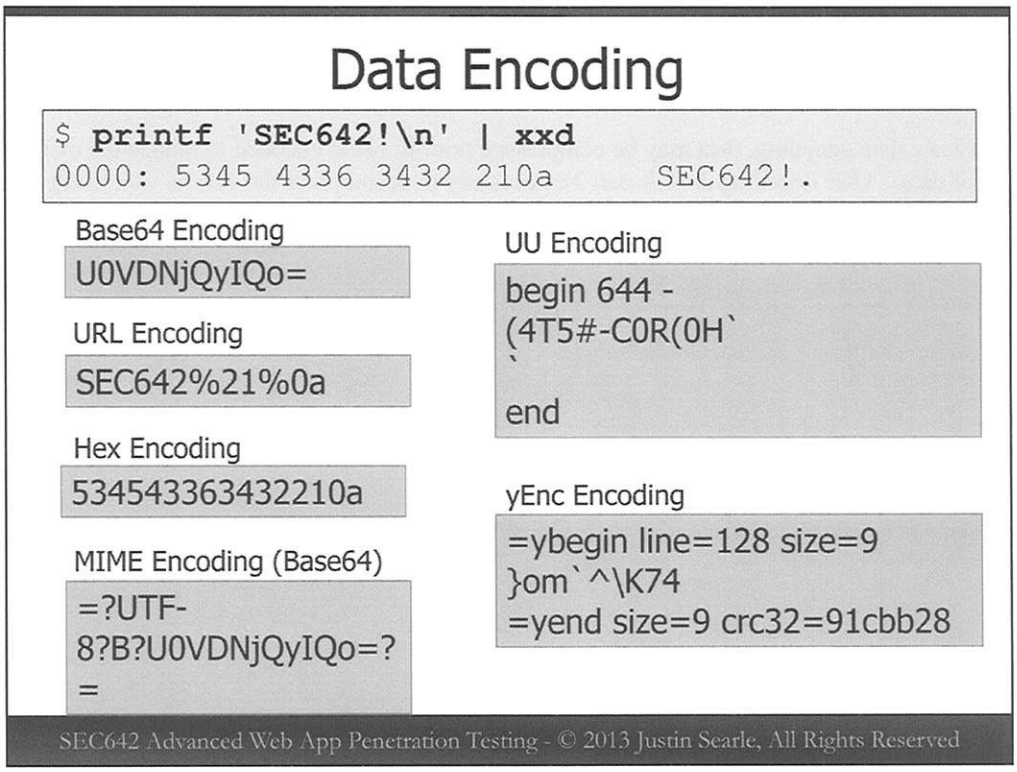


Identifying Cryptography

The question shown at the top of this slide was taken from the SANS Pen-Test Curriculum GPWN mailing list (https://lists.sans.org/mailman/listinfo/gpwn-list). When we examine cryptography in web applications, we typically have a lot more questions than we might find answers for, such as:

- For a given string, what encryption or hashing function was used to generate a known output?
- For an encrypted value, what algorithm was used to protect the plaintext?
- Is the observed data set encrypted, hashed or obfuscated content?

There are some opportunities to answer these questions through the analysis of data, commonly relying on large data sample sets while analyzing predictability, entropy and other factors.



Data Encoding

Since HTTP is a 7-bit ASCII protocol, we need a mechanism to encode 8-bit data into an ASCII form. Further, many ASCII characters are reserved in HTTP and HTML, requiring an added level of abstraction for multiple characters.

Data encoding is at the developer's discretion, although it is commonly influenced by the capabilities of the web application framework used. Most commonly, binary data is Base64 encoded, as shown on this slide, using 64 characters to encode any 8-bit value. Base64 systems can vary, but commonly leverage uppercase and lowercase characters and numbers (representing 62 characters) as well as the "+" (plus) and "/" (forward slash) characters.

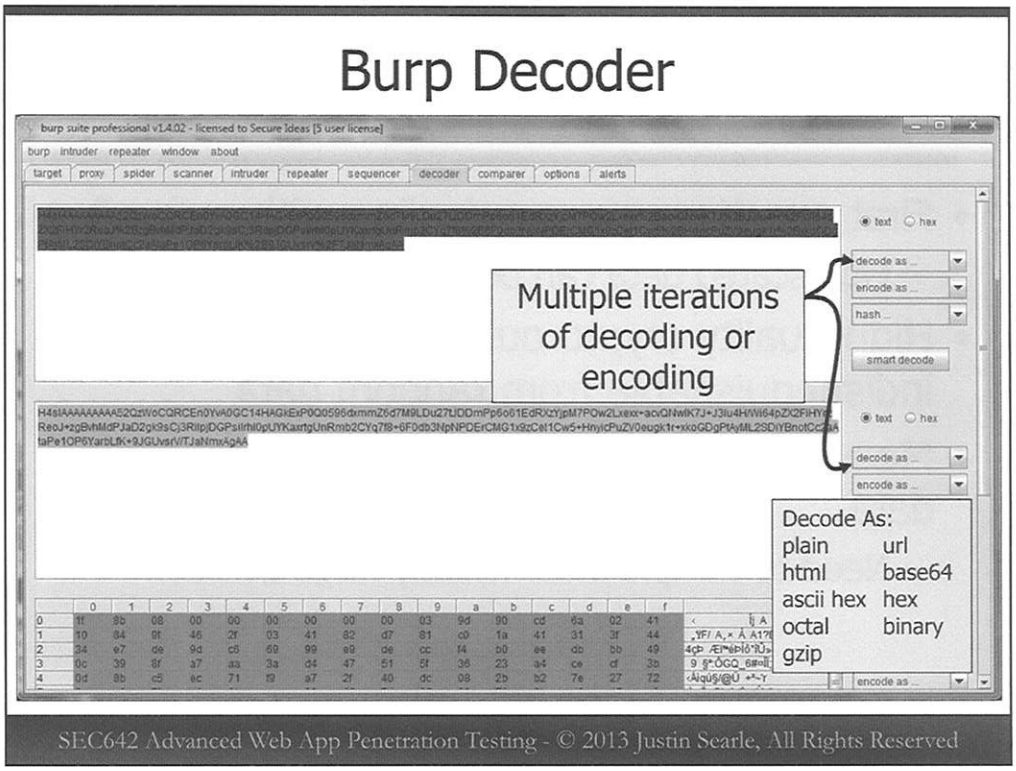
URL encoding (also known as percent encoding) is also commonly used in web applications, conforming to the rules of RFC3986. In URL encoding systems, reserved ASCII characters ("! * ' ( ) ; : @ & = + $, / ? # [ ]") are represented in their hexadecimal value with a leading percent sign. Similarly, binary data can be represented in a similar fashion, as shown on this slide.

Hex encoding is also commonly observed where each character, regardless of whether it is ASCII or binary in nature, is represented with a 2-byte hexadecimal ASCII value.

Multipurpose Internet Mail Extensions (MIME) is a mainstay protocol of the Internet, commonly used for e-mail and other protocols to embed ASCII content representing binary data. One of the two MIME encoding formats includes Base64 encoding with additional header and trailer structure.

Additionally, UU encoding was leveraged widely for early Internet protocols such as UUCP, but has largely been deprecated from widespread use. Similarly, the yEnc encoding protocol represented an opportunity to reduce the overhead from MIME and UU encoding formats, but was never widely leveraged.

In addition to basic data encoding, data may be compressed prior to being encoded to reduce the overhead for the transmission of data. After decoding data, it may be necessary to decompress the data as well using common or proprietary compression schemes.



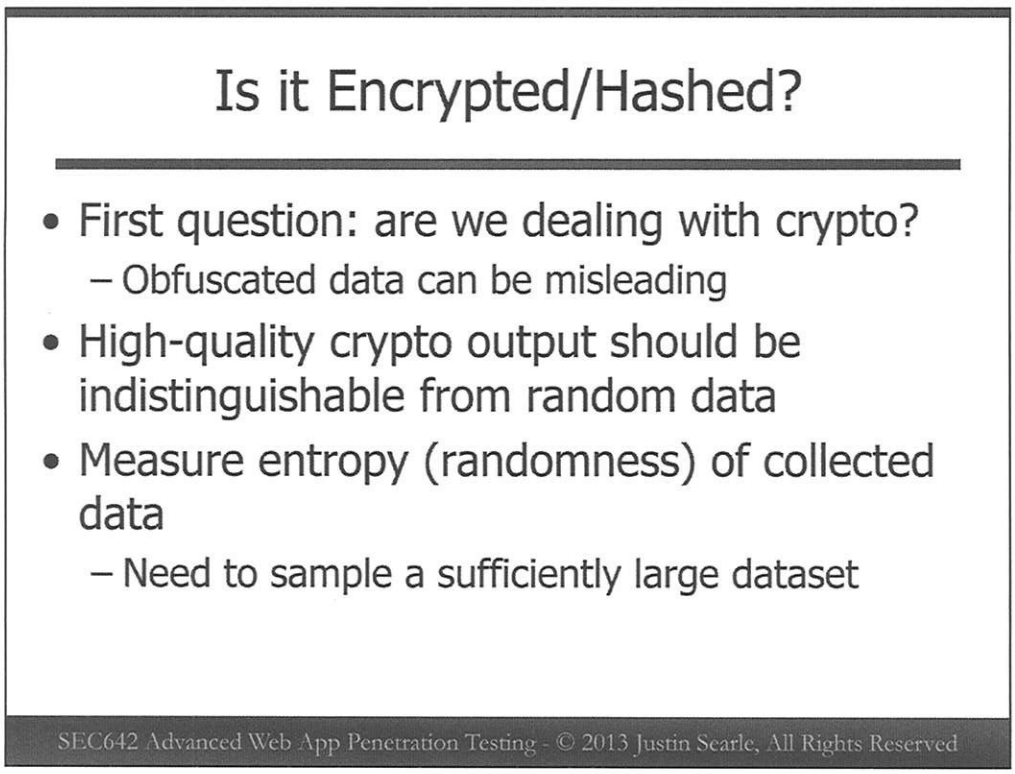
Burp Decoder

Burp Suite is a sophisticated and powerful framework for evaluating the security of web applications. We will leverage Burp throughout the course for the evaluation of web application functionality.

Burp includes a feature known as Burp Decoder which gives us a flexible opportunity to apply decoding rules to data. For any selected data, right-click the data and select "send to decoder". In the decoder tab, we can take the initial data and easily experiment with decoded formats, selecting the option to decode as:

- plain: Apply no decoding
- HTML: View the HTML source for the data
- ASCII Hex: View the data in 2-byte ASCII values representing the input hex
- Octal: View the data in 7-bit Octal format
- Gzip: Decompress the data from Gzip-compressed data
- URL: Apply URL-decoding rules (removing percent-encoding)
- Base64: Decode Base64-encoded data
- Hex: Show the data in hexdump format
- Binary: Show the data in a binary representation

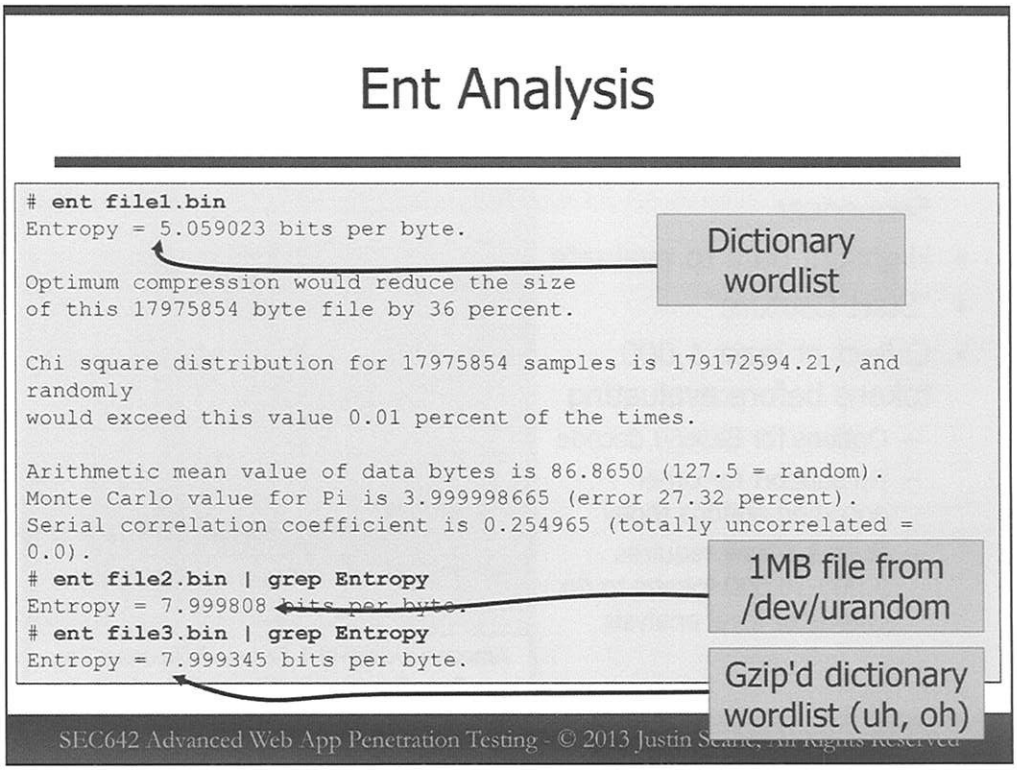
With Burp Decoder, we can apply multiple decoding passes easily. As an example, this gives us the ability to take an initial data set, decode using URL-decode, then Base64 decode, and then Gzip decode to get to the original data.



Is it Encrypted/Hashed?

When dealing with unknown data, we will commonly need to decide if the data is encrypted or hashed. Even if the data appears to be random without a prevalent number printable ASCII characters, it does not necessary indicate that the data is encrypted.

One of the desirable goals for encrypted data is that encrypted content should be indistinguishable from random data. We can use this property to our advantage to identify encrypted content by measuring the entropy or randomness of the data. In order to measure entropy, however, we need to collect a sufficiently large dataset of hundreds to thousands of bytes, possibly requiring the development of a short script to retrieve, decode and append data to a dataset that can be analyzed.



Ent Analysis

Ent is a simple command-line tool to measure the entropy of a given file. Ent can be used to measure the entropy of data to determine if the data has high entropy and therefore could be encrypted content.

In order to use Ent to analyze content from an HTML cookie, we would need to develop a short script to retrieve the cookie content, decode the data as necessary to return it to the natural binary form, and save it to a file. This process may have to be repeated multiple times to obtain hundreds to thousands of bytes of data to be evaluated.

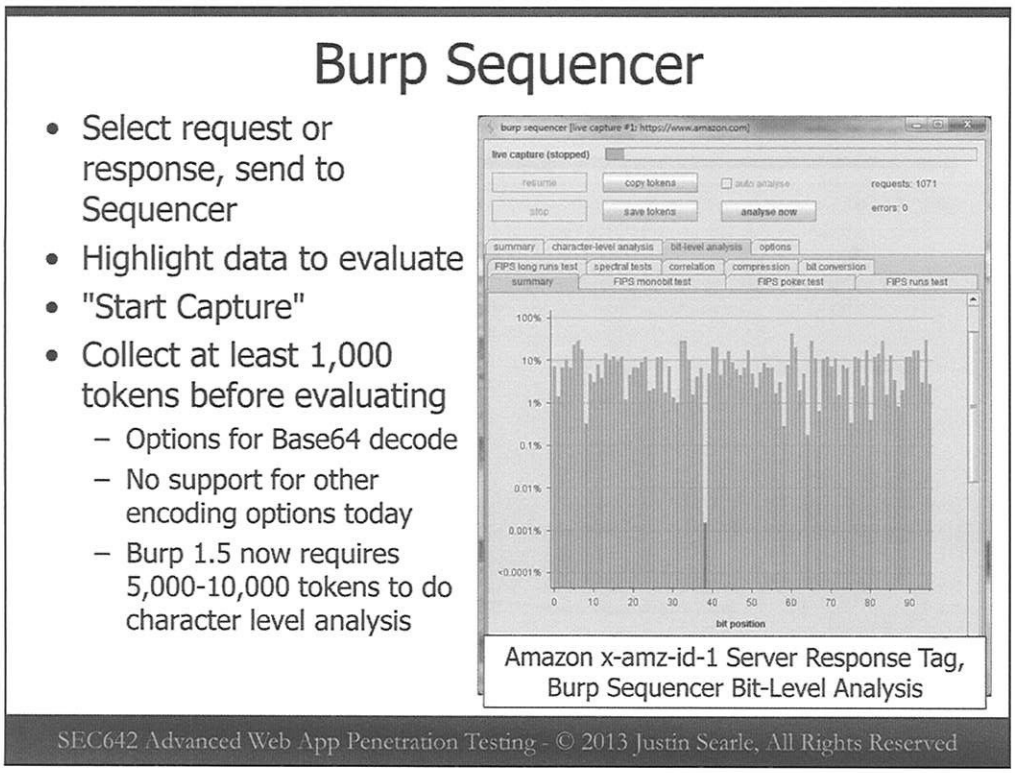
To use Ent, simply invoke the ent command-line tool with the filename to be reviewed, as shown on this slide. In the first example, we have Ent review the entropy of an ASCII dictionary file where Ent reveals the entropy of the file as 5.05 bits per byte, a relatively low entropy value.

Next we ask Ent to review the entropy of a 1 MB file read form the Linux /dev/urandom device. In this case, Ent reports a very high entropy level of 7.9998 bits per byte, indicating a high level of entropy or file randomness.

Finally, we ask Ent to review the same wordlist in the first example, except now the wordlist has been gzip compressed. Compression functions make a file smaller by removing the repetition from the file and, therefore, changing the entropy of the file itself. In this test, the compressed wordlist file has a very high entropy level of 7.9993 bits per byte, nearly that of the data read from the Linux random number device.

Ent can be a valuable tool for evaluating the content of collected data, but the results can also be misleading unless the content is carefully scrutinized first. For most web application assessments, we'll want an interface that can apply multiple tests to evaluate the randomness and entropy of data.

Ent is the excellent work of John Walker, available at http://www.fourmilab.ch/random.

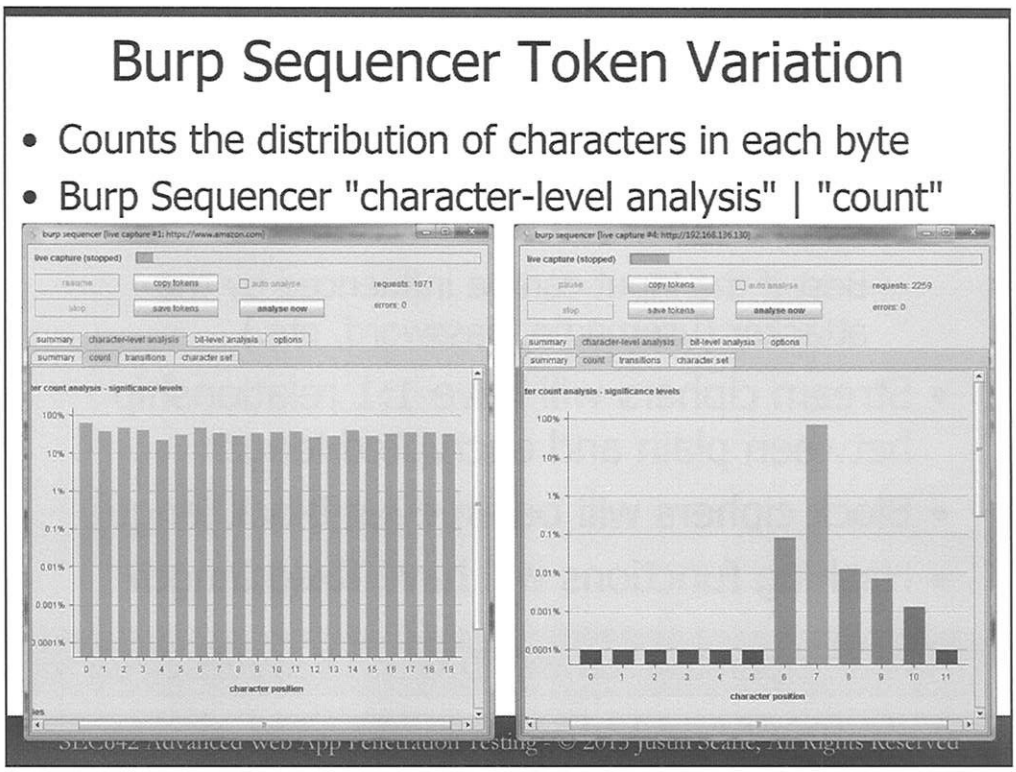


Burp Sequencer

Burp's Sequencer feature allows us to quickly collect data from a target web system and apply several randomness quality checks. From any Burp Proxy request or response, right-click on the item and select "send to sequencer".

In the Sequencer view, highlight the data to evaluate while using Burp's offset/regular expression matching tool to ensure the desired data is evaluated. Next, select "Start Capture"; Burp will send the original request multiple times, each time obtaining and recording the marked data for analysis from the response. After collecting at least 1,000 data samples you can analyze the data, decoding the original content from Base64 to the natural value in the options tab if necessary.

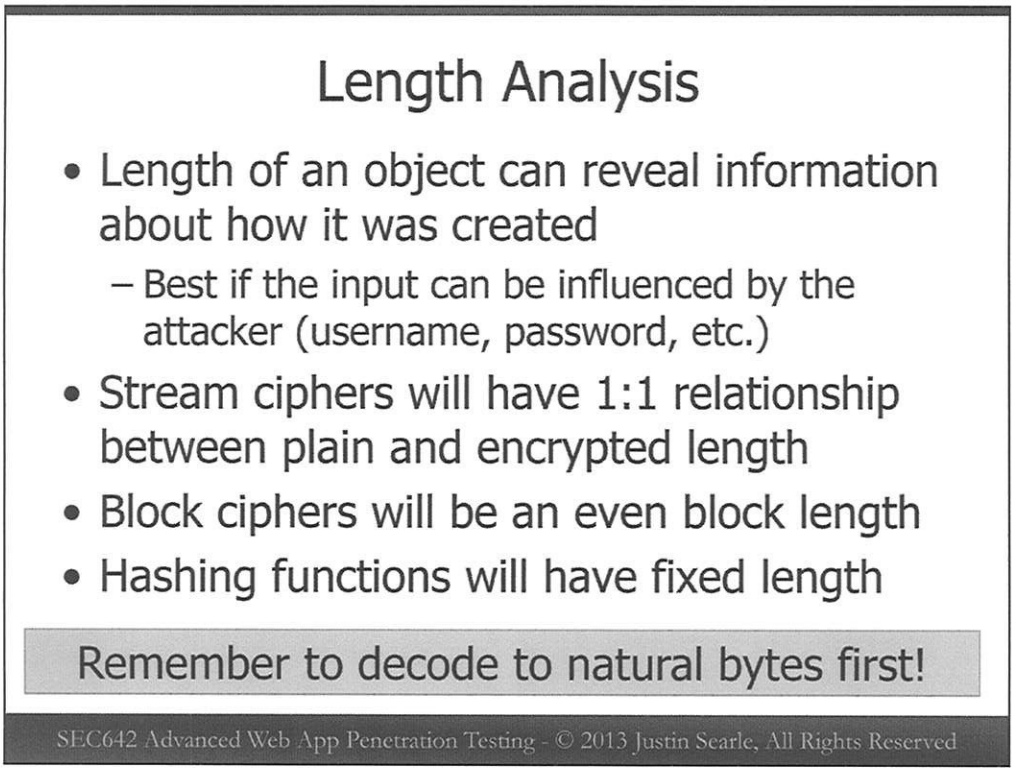
The example on this slide shows the bit distribution of the Amazon.com x-amz-id-1 server response tag. While the content appears to have a high level of entropy, several fields are non-uniformly distributed (such as the value near bit position 38), possibly indicating that this field or a portion of this field is a bitmask of some sort used for tracking information as opposed to a purely randomized value.



Burp Sequencer Token Variation

Burp's Sequencer applies many tests to evaluate the randomness of the retrieved content, however, the character-level analysis of the data represents a very useful analysis metric for identifying the randomness of the data. In the example on this slide, two data sets were samples. On the left, the variation in the count of each unique byte value is minimal, where no single byte value was more random or more static than the others (by a wide margin). By contrast, the example on the right shows that the first 6 bytes were static, while bytes 7-11 showed a higher level of variation.

From this analysis we can see that the data on the right is not encrypted due to the lack of entropy in the data. It is not clear if the content on the left is encrypted (a possibility) or if it is randomly selected.



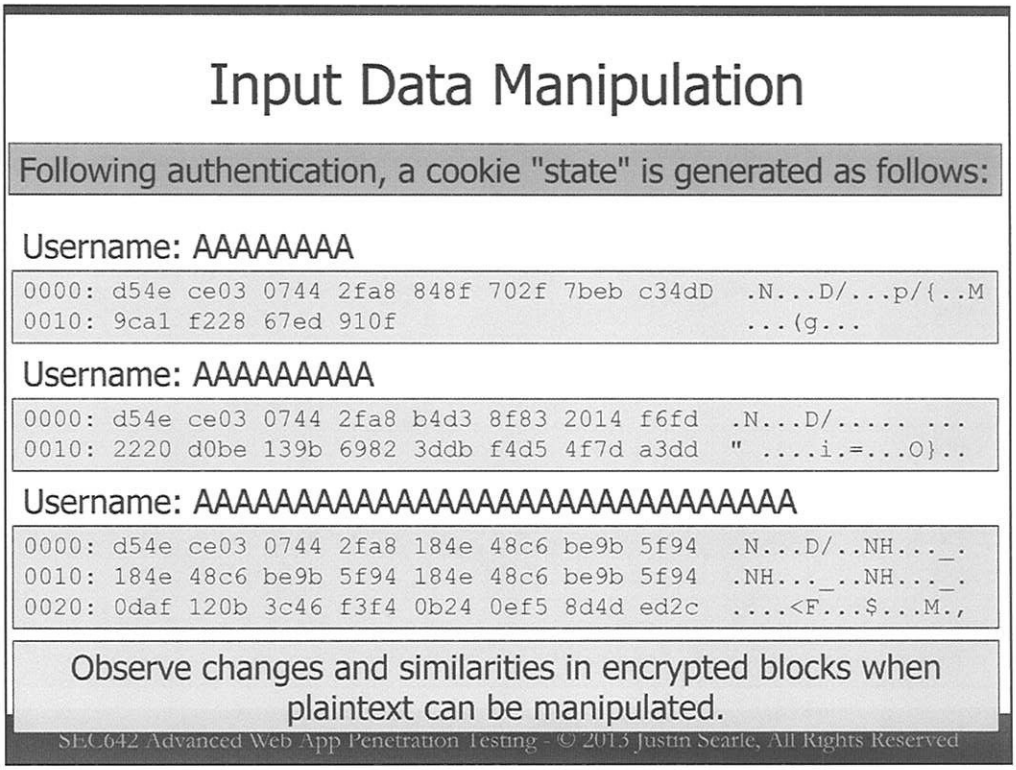
Length Analysis

In addition to entropy analysis, evaluating the length of a blob of data can be a useful mechanism to identify encrypted or hashed content. Length analysis works best when we can also influence the content that is encrypted or hashed to observe an increase or decrease of the content length while we increase or decrease the length of our input values.

Recall that stream ciphers will have a 1:1 relationship between the plaintext and the encrypted length. The ciphertext length may not be the same as the plaintext (due to the presence of headers, other unknown fields or checksum data), but it will grow in length by an equal number of bytes as our input length changes.

By contrast, a block cipher will always have an even block length, regardless of the length of the input data. Hashing functions will have a fixed length value, regardless of the input data length.

Always remember to decode the data to be assessed into the natural byte values before reviewing the length of data. For example, if the data is Base64 encoded, be sure to Base64 decode the data prior to counting and evaluating the length of the data.



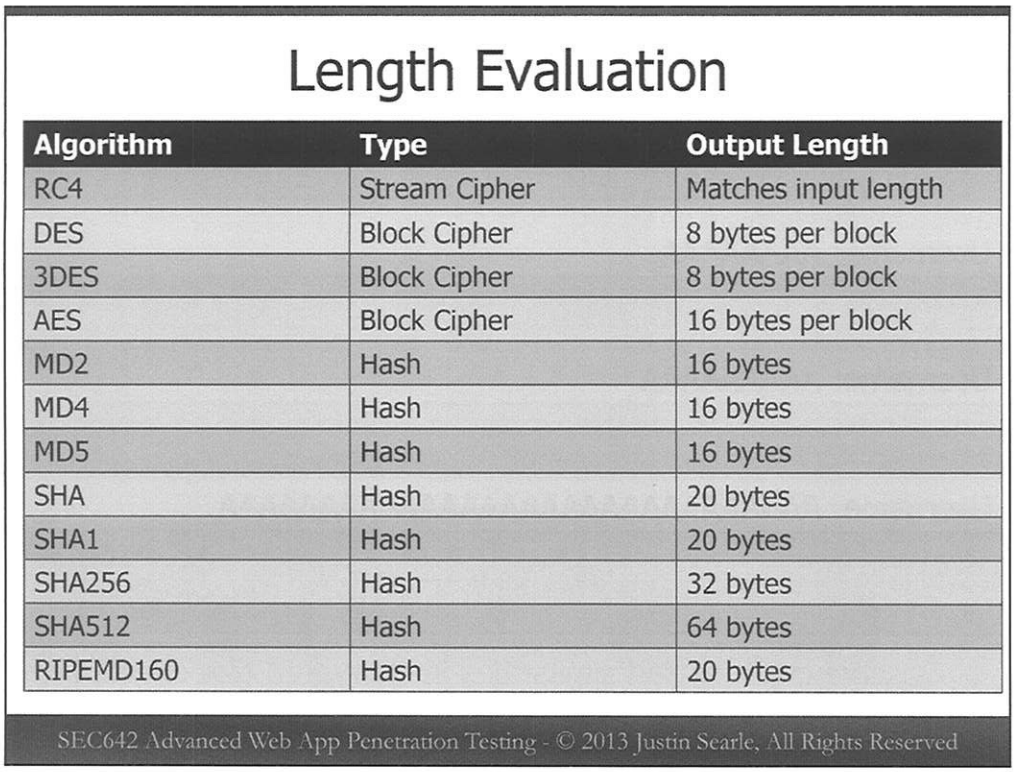
Input Data Manipulation

Consider a case where a cookie value called "state" is generated as a component in a web application. The hex output in the first example is the decoded cookie content when a username of "AAAAAAAA" is used. The presence of three 8-byte blocks indicates that the data is possibly encrypted with a block cipher mechanism.

The second example is generated with a username of 9 "A" characters (one longer than the previous example). In this output, we jump from 3 ciphertext blocks to 4 ciphertext blocks even though we only added one character to the input value. This helps us confirm that a block cipher is used for this content.

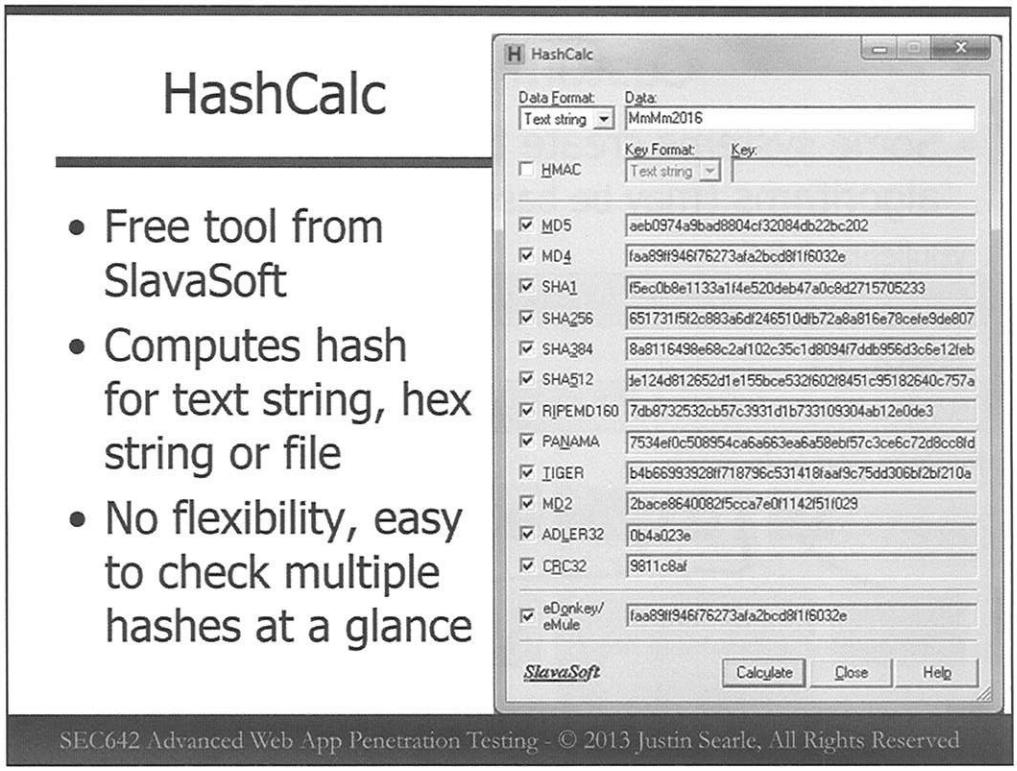
The third example uses a much longer username of 30 "A" characters. The data blob grows another 2 blocks (consistent with an 8-byte block cipher). Further, we can see some duplicate encrypted blocks present, indicating the use of ECB mode to encrypt the data.

Evaluating the length of data is one of the most useful mechanisms for evaluating the content of data and identifying the presence of encrypted or hashed data. We get additional benefit when we can also manipulate the input to the data as well.



Length Evaluation

This slide lists several encryption and hashing algorithms, along with the type of the algorithm and the output length value. Keep this reference handy when evaluating data to help identify the algorithm used.



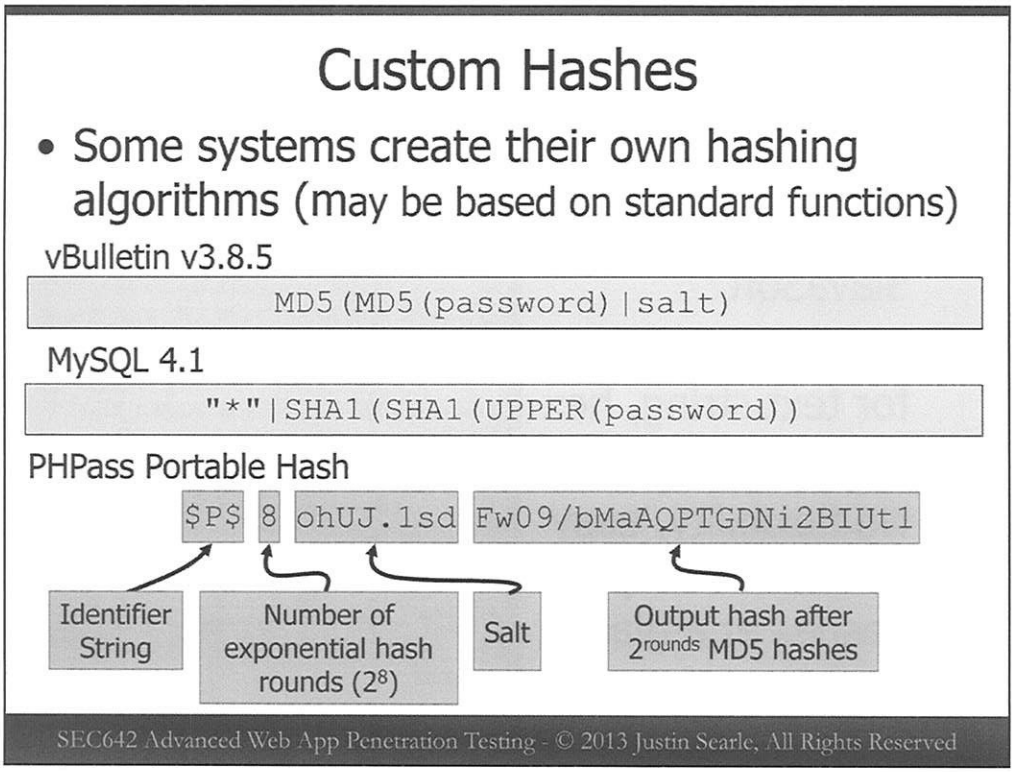
HashCalc

HashCalc is a free, simple tool for Windows systems from SlavaSoft used to compute several hash values for a given string or input file. HashCalc allows you to hash an input value in several methods and compare the output result to the observed hash to identify any matches.

In the beginning of this module we displayed a post from the GPWN mailing list:

"I need you help to figure out what hash function produce 64 bit output and produce this hash 2F8451C95182640C757A for the string MmMm2016?"

On this slide we see that, for the input string MmMm2016, the hash value reported is the last 10 bytes of a SHA512 hash. It is common for developers to take a portion of an output hash to reduce the storage space needed to record the hash while adding a minor level of obscurity to the hashing function.



Custom Hashes

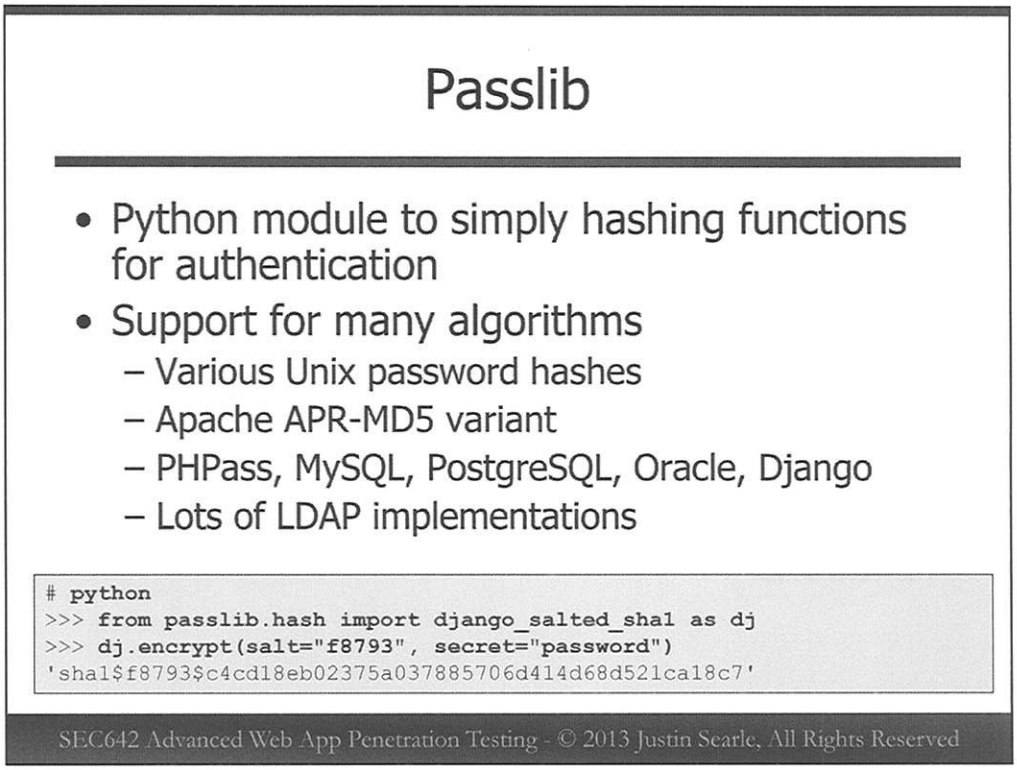
While HashCalc will display several standard hash values for the input content, it does not give us visibility into custom hashing functions. Many web application developers will create their own hashing functions to suit their needs for a given application.

For example, the popular vBulletin software uses a custom hashing function for password storage by MD5 hashing the password, concatenating the salt to the inner hash, and then MD5 hashing the value again (as of vBulletin 3.8.5).

MySQL 4.1 also has a similar mechanism, double-SHA1 hashing the password after converting it to uppercase, then pre-pending a static asterisk ("*") character to the hash result.

PHPass Portable Hash function concatenates a salt to the session value, then hashes it for a variable number of times, depending on the value noted in the first byte following the identifier string "P".

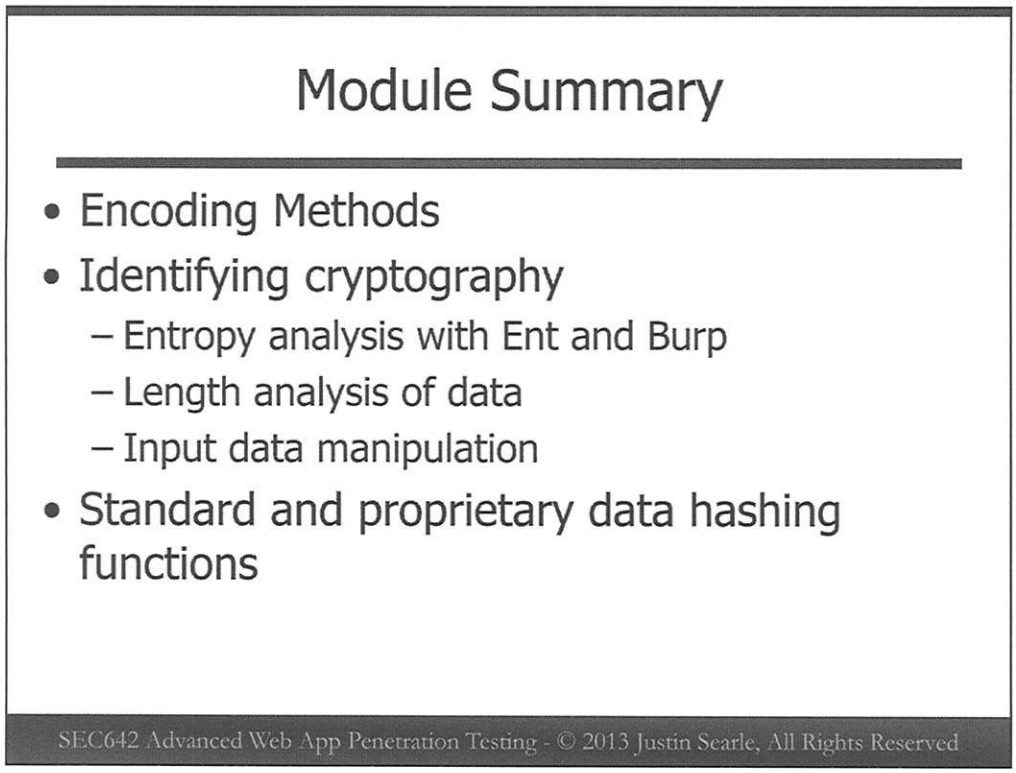
It isn't necessary to memorize the hashing function for each unique application design, but it is important to recognize that applications may vary, and it will be difficult in many cases to ultimately identify exactly how hashes are generated. Fortunately, many applications are using standard notation mechanism to differentiate their hashes from other application hashes, such as the leading "*" for MySQL and the "P" for PHPass.



Passlib

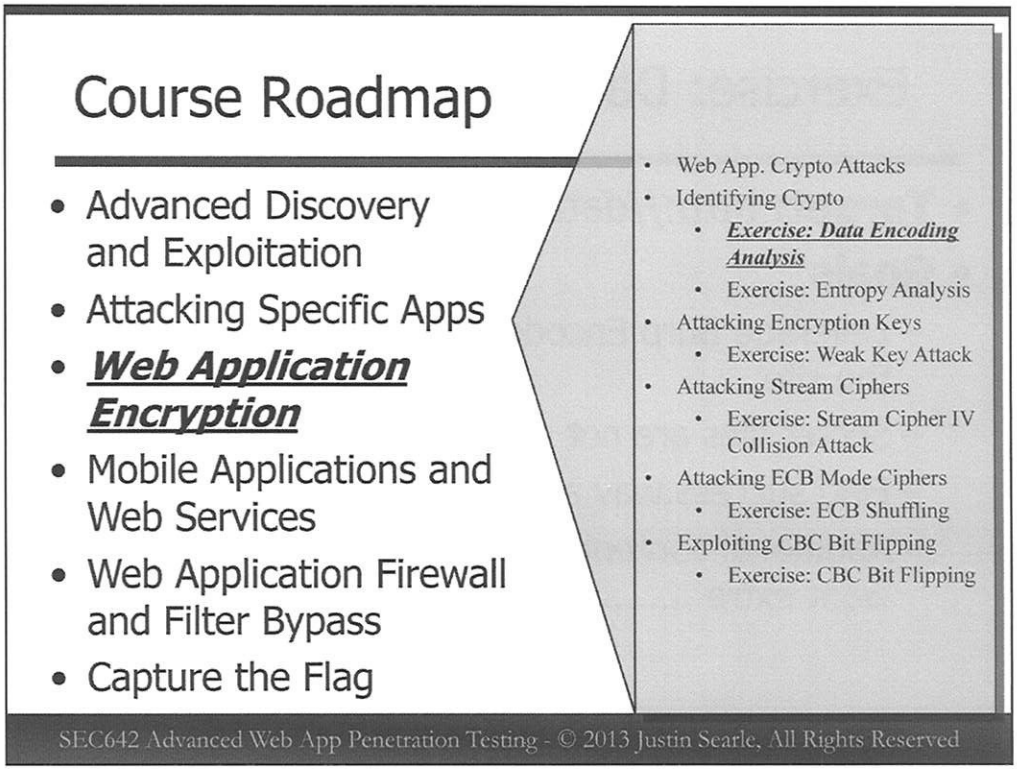
Passlib is a Python module used to create password hashes in a platform-independent function simply. Using Passlib, we can specify the input value and a salt value (when required) to generate an output hash for several proprietary systems including various Unix password hashes, Apache APR-MD5, PHPass, MySQL, PostgreSQL, Oracle, Django and more.

Passlib can be retrieved from http://packages.python.org/passlib/lib/passlib.hash.html.



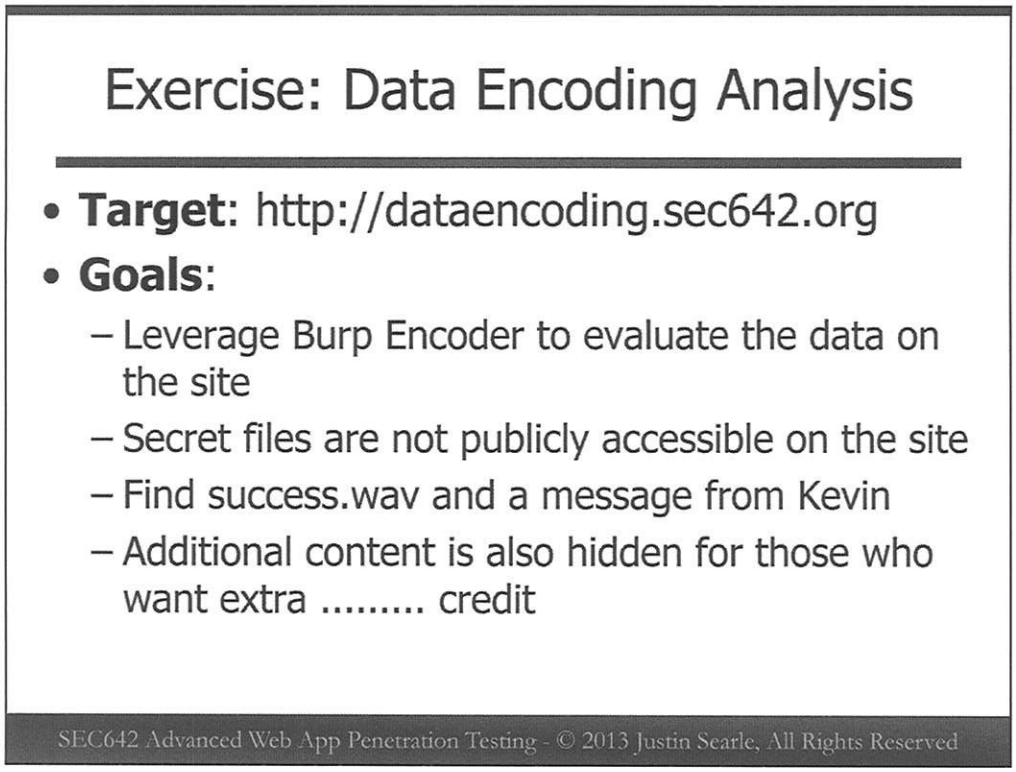
Module Summary

In this module we looked at techniques to identify cryptography including entropy analysis, length analysis and input data manipulation analysis. We examined the many encoding options available that we will encounter in web app assessments, leveraging Burp's flexible decoder functionality to decode the content into a native data form. Finally, we examined multiple standard and proprietary data hashing functions, looking at tools we can use to generate the hashes to compare known input to an observed hash value.



Course Roadmap

Now we will do an exercise.

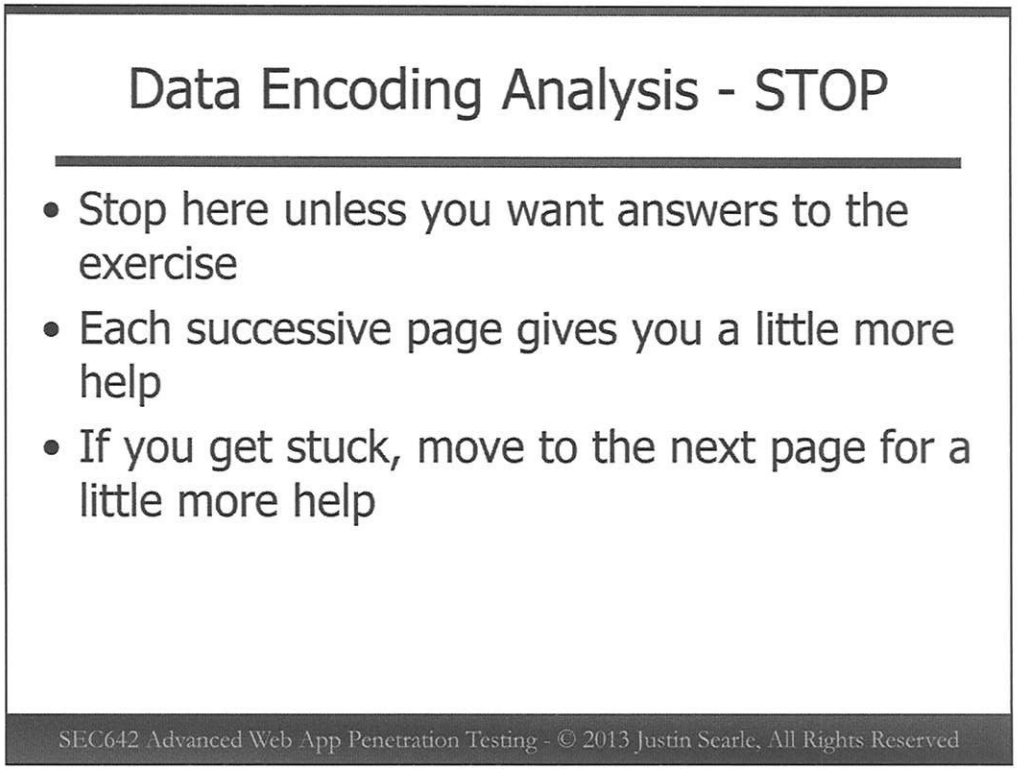


Exercise: Data Encoding Analysis

In this exercise you are tasked with evaluating a target web application at http://dataencoding.sec642.org. For this exercise you will need the Burp tool for the Encoder functionality, which you can retrieve from http://files.sec642.org. You can run Burp on any platform you wish.

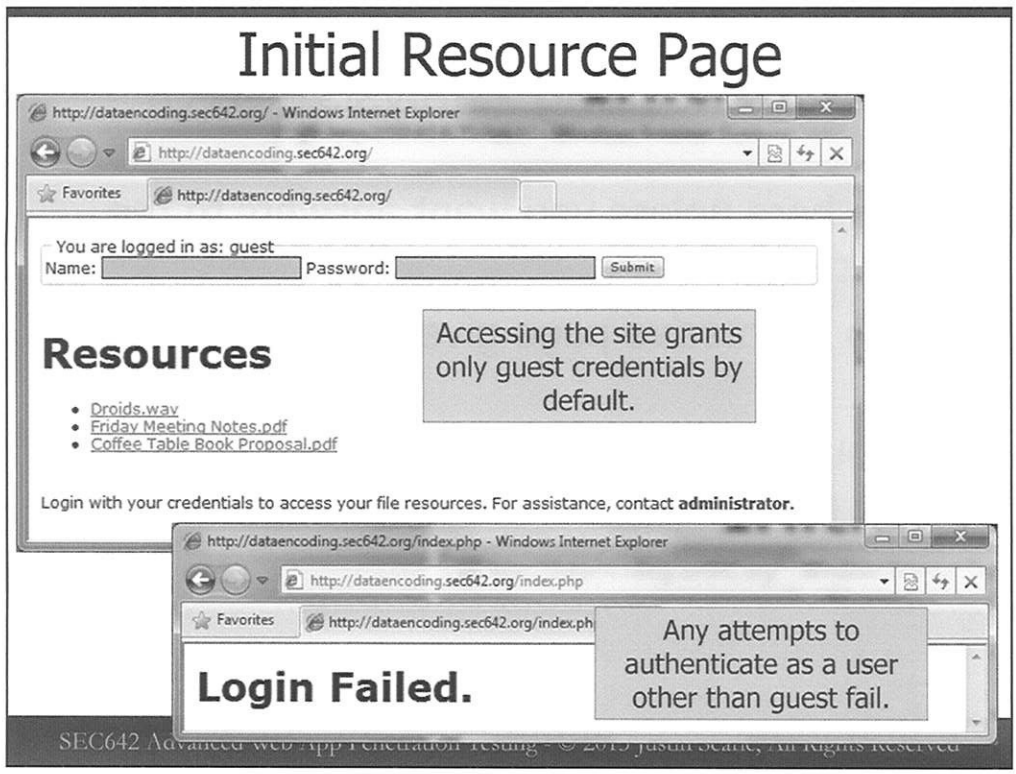
On the target site you have been informed that there are some secret files that are intended for authenticated users. One file, "success.wav" is your target. Use your skills as an analyst to retrieve this file and listen to the content.

Other files are also hidden on the site for those who complete early and want extra credit.



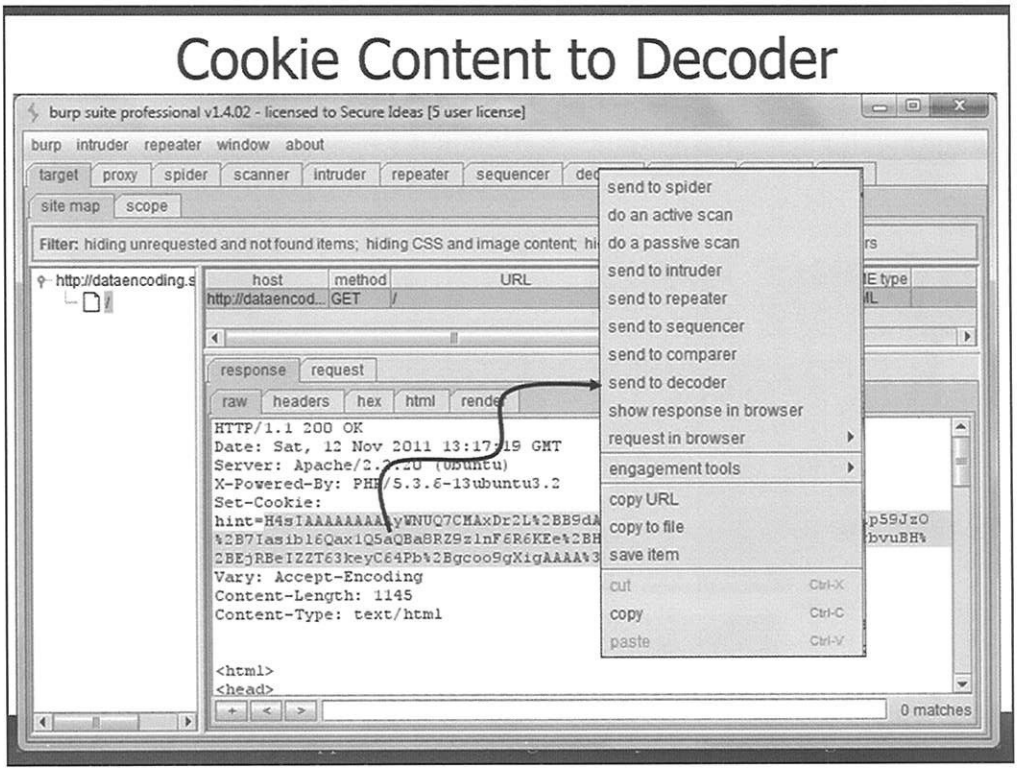
Data Encoding Analysis – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



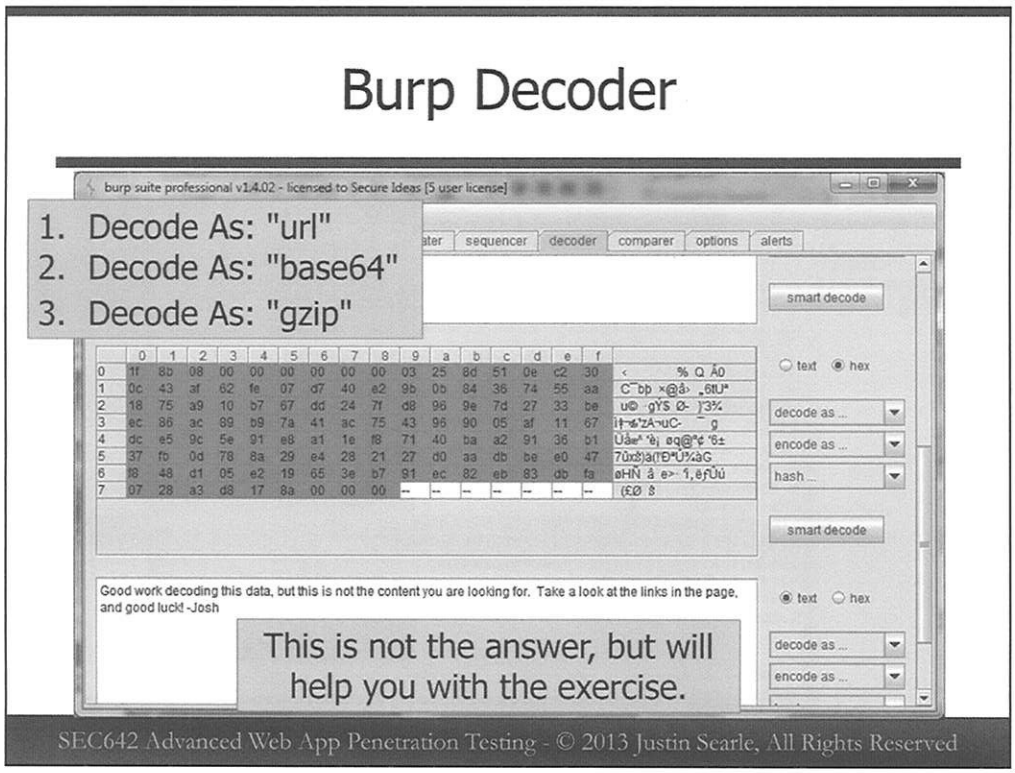
Initial Resources Page

Browsing to the site reveals a limited number of resources for the guest user account. A username and password dialog is also present, but all requests to authenticate as a user fail. You must assess the site with only the privileges of the guest user.



Cookie Content to Decoder

Examining the page request with Burp we see that a cookie of "hint" is set when we access index.php. Highlight the cookie content to the right of the equal-sign after the cookie name, then right-click on this content and select "send to decoder".



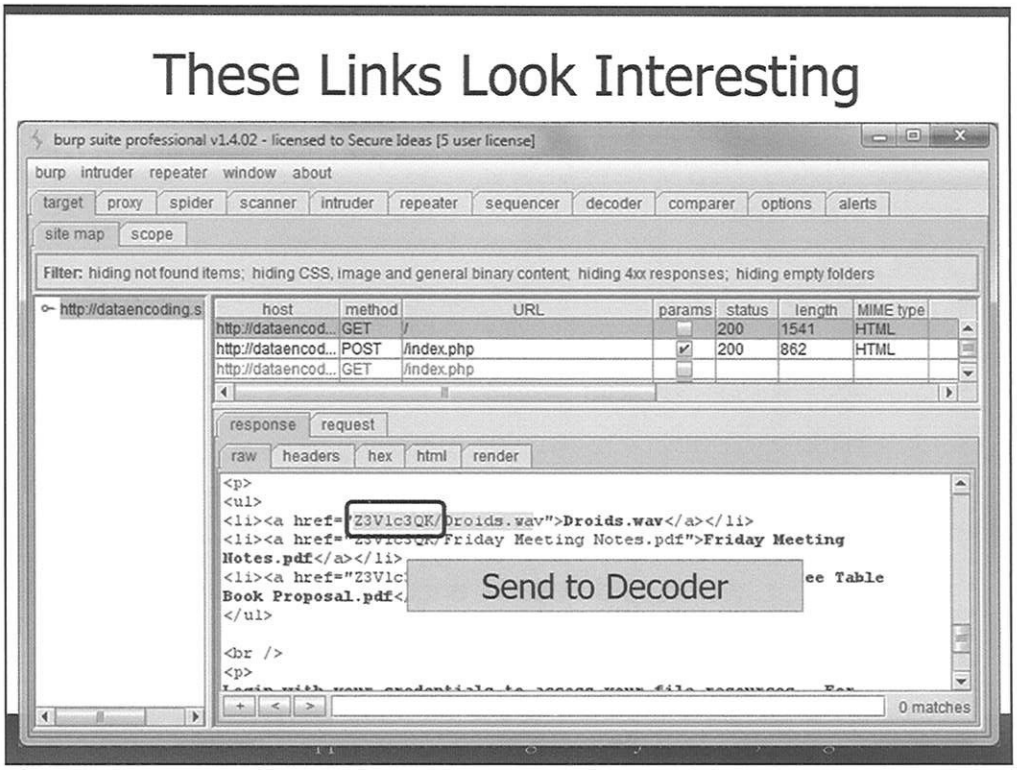
Burp Decoder

Experimenting with the Burp Decoder options, we need to decode the cookie content as follows:

1. URL Decode
2. Base64 Decode
3. Gzip Decode

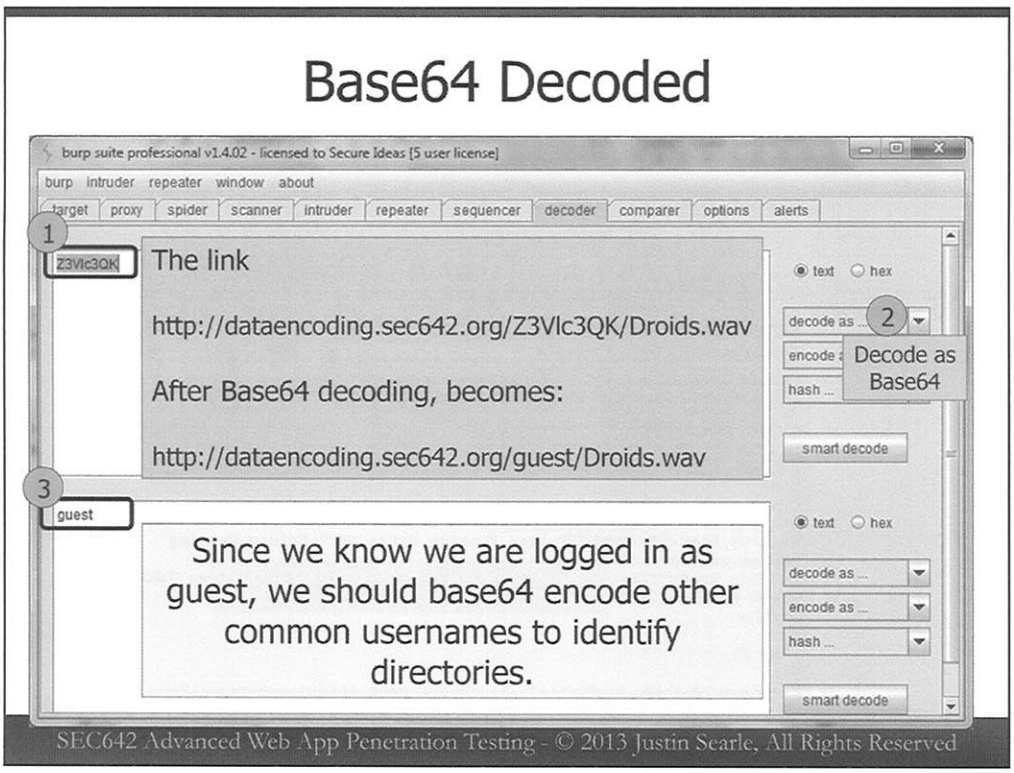
We know that the content is URL encoded by seeing the percent-encoding symbols in the output (the "%" character is not permitted in Base64 encoding). Burp will attempt to Base64 decode if you tell it to do so prior to URL decoding but this will generate invalid output data.

The plaintext after gzip decoding gives you a hint to complete the exercise.



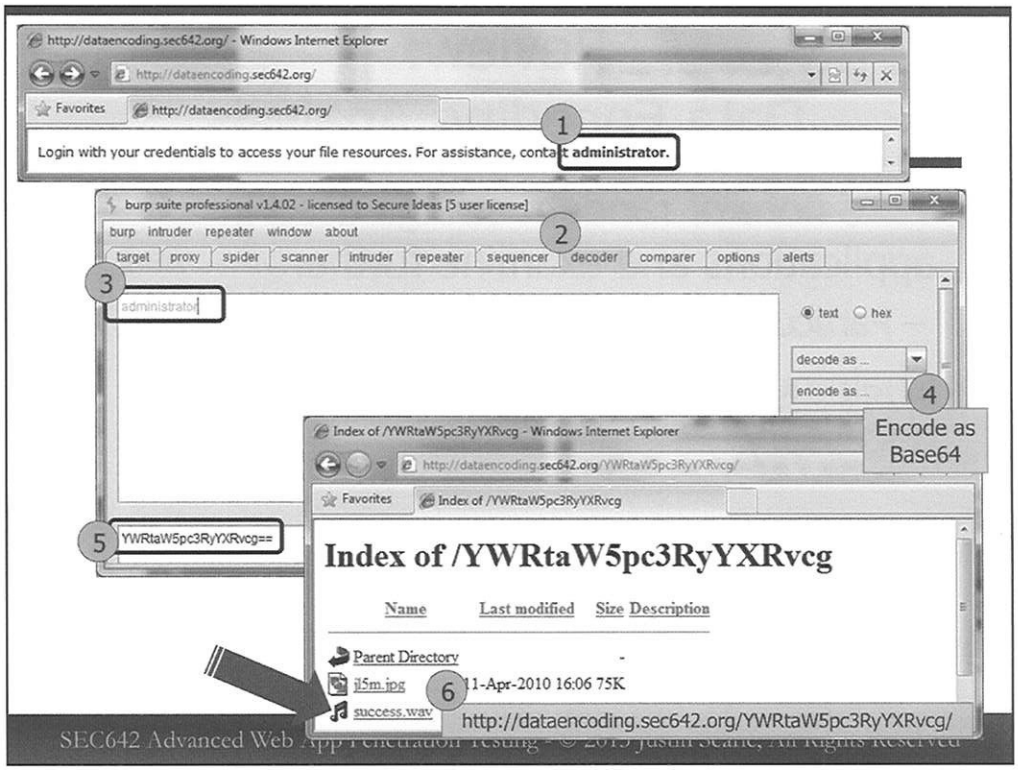
These Links Look Interesting

Looking at the links on the main page for guest-level files, we see names such as "Z3Vlc3QK/Droids.wav". This looks like an attempt to create unpredictable directories to keep content private. Highlight the "Z3Vlc3QK", right-click and send to decoder.



Base64 Decoded

After Base64 decoding the "Z3Vlc3QK" content, Burp reveals the plaintext as the string "guest". Since we are logged in as guest, we can ascertain that the private directory system is simply a Base64 encoded version of the username. Since we are logged in as guest, we should try to encode other common usernames to identify other directories on the file system and attempt to access them.



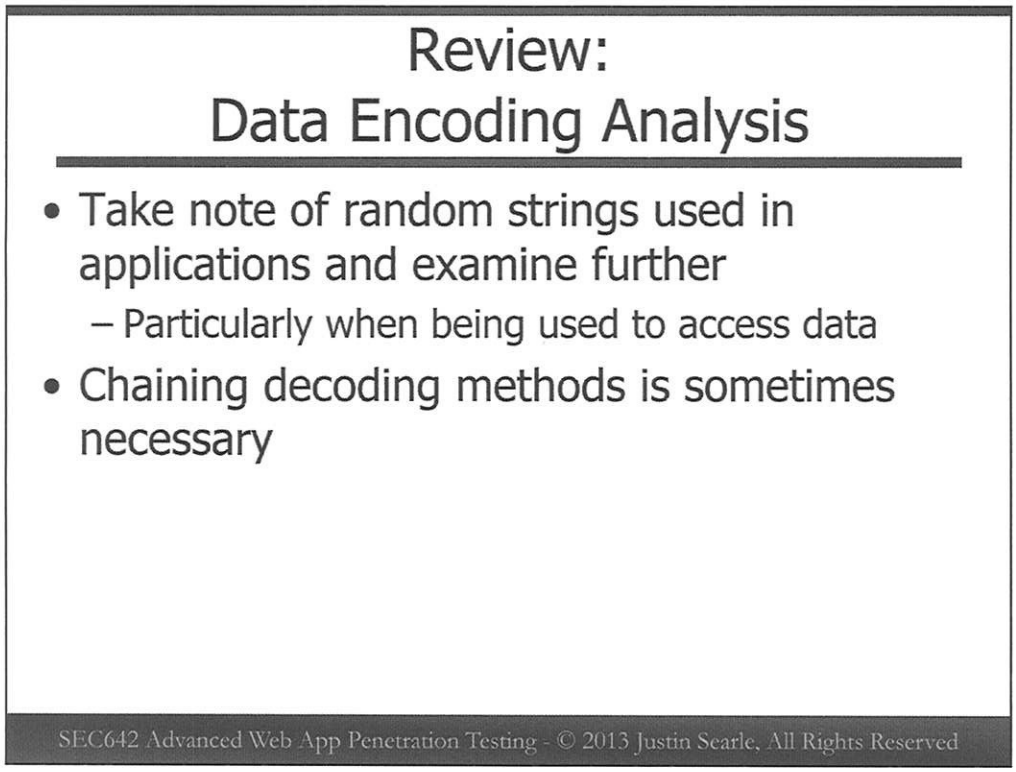
Administrator User

On the index page for the site, a note tells the user that if they need help with the site they should contact "administrator" (in bold). This is a username to test with to see if there are files present for this user.

Navigate to the decoder tab and enter the name "administrator". Encode the string in base64 format to observe the content "YWRtaW5pc3RyYXRvcg==". Then, try to browse the to site with the following URLs:

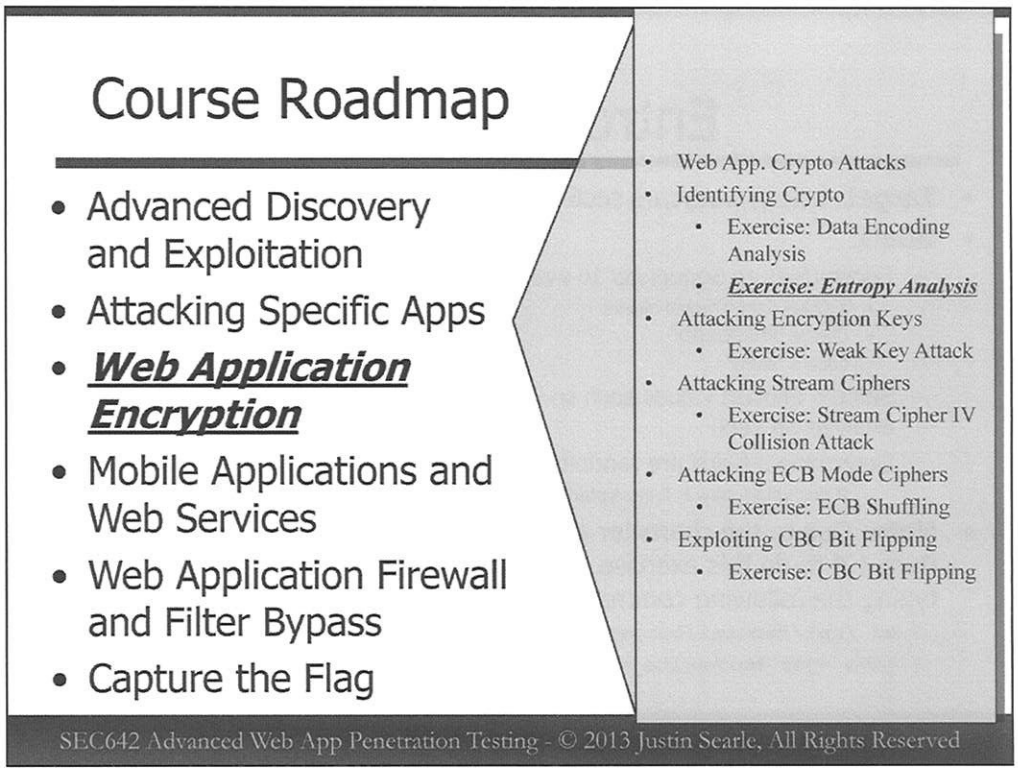
- http://dataencoding.sec642.org/YWRtaW5pc3RyYXRvcg==
- http://dataencoding.sec642.org/YWRtaW5pc3RyYXRvcg

While the first URL returns a 404, a modification to the Base64 string by removing the equal sign characters (the equal sign is a padding character in Base64 to make the string a number of 4-byte blocks) grants access to a browseable directory, granting access to the success.wav target file.



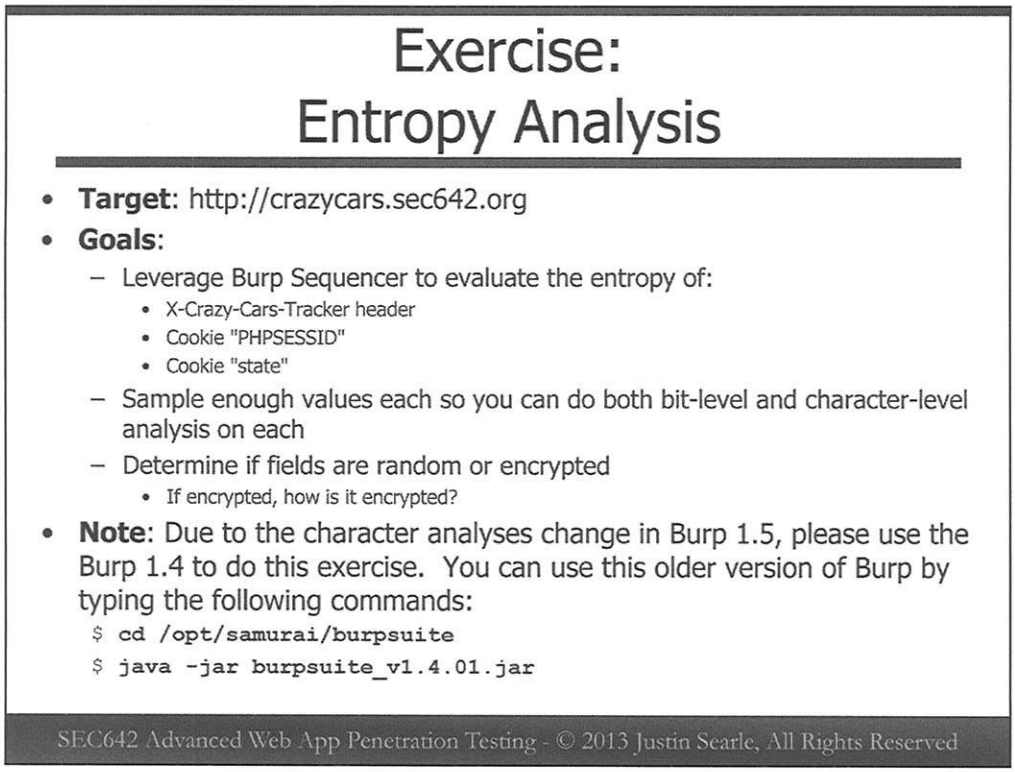
There are two main points to this exercise that we need to keep in mind. First, we need to watch for random appearing values and how they are used in our target web applications. This is especially true when a random value is used to access data or a resource. The base64 encoded value in the URL could have been truly random and used in a query to find the required file. Or the developer may have written the application to simply decode the value and then access the requested resource. We need to catch it when it is done incorrectly, such as in this exercise.

Last, it may become necessary to chain different methods of decoding data together to find the source data we are looking for. In this exercise we had to decode the "hint" cookie from URL, base64 and gzip encoding. Burp makes working with this process fairly easy with the Decoder. Become familiar with what the different encoded data formats look like to assist you in selecting your decoding methods.



Course Roadmap

Now we will do an exercise.



Exercise: Entropy Analysis

In this exercise you are to evaluate the entropy of several tokens from the target website at http://crazycars.sec642.org. We'll return to this website for additional exercises throughout the material.

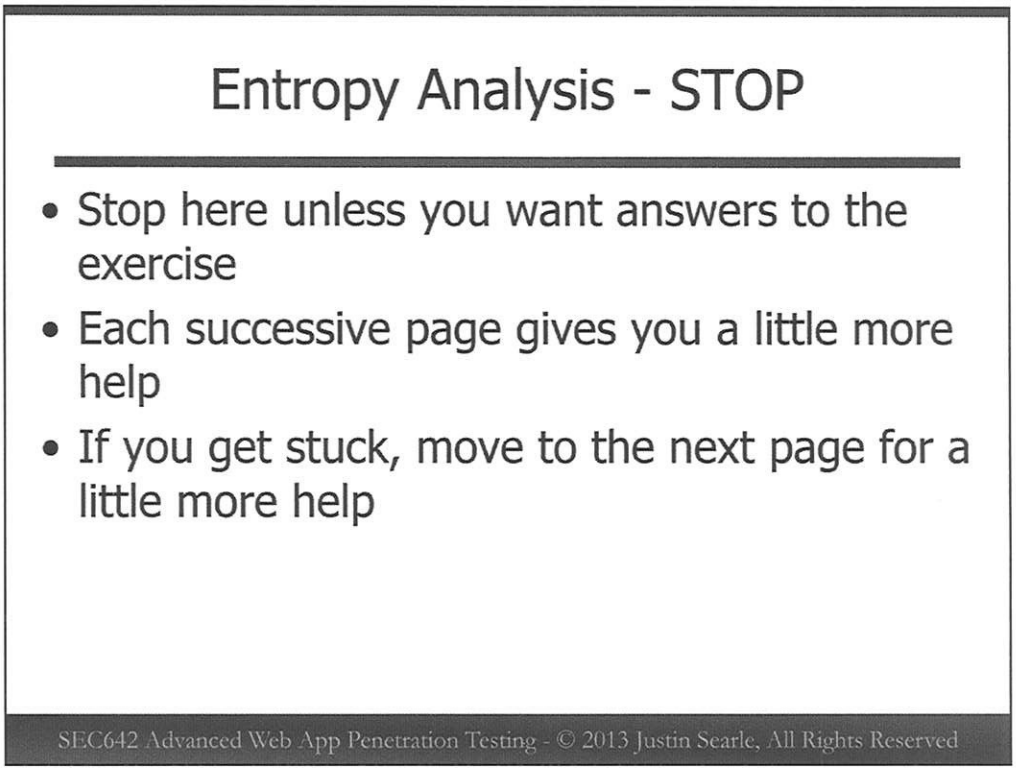
Using Burp Sequencer and Encoder, evaluate the entropy of three tokens:

- X-Crazy-Cars-Tracker header
- Cookie "PHPSESSID"
- Cookie "state"

For each token, sample approximately 500 values each. Determine if the sampled fields are random or encrypted. If there are fields that may be encrypted, determine how.

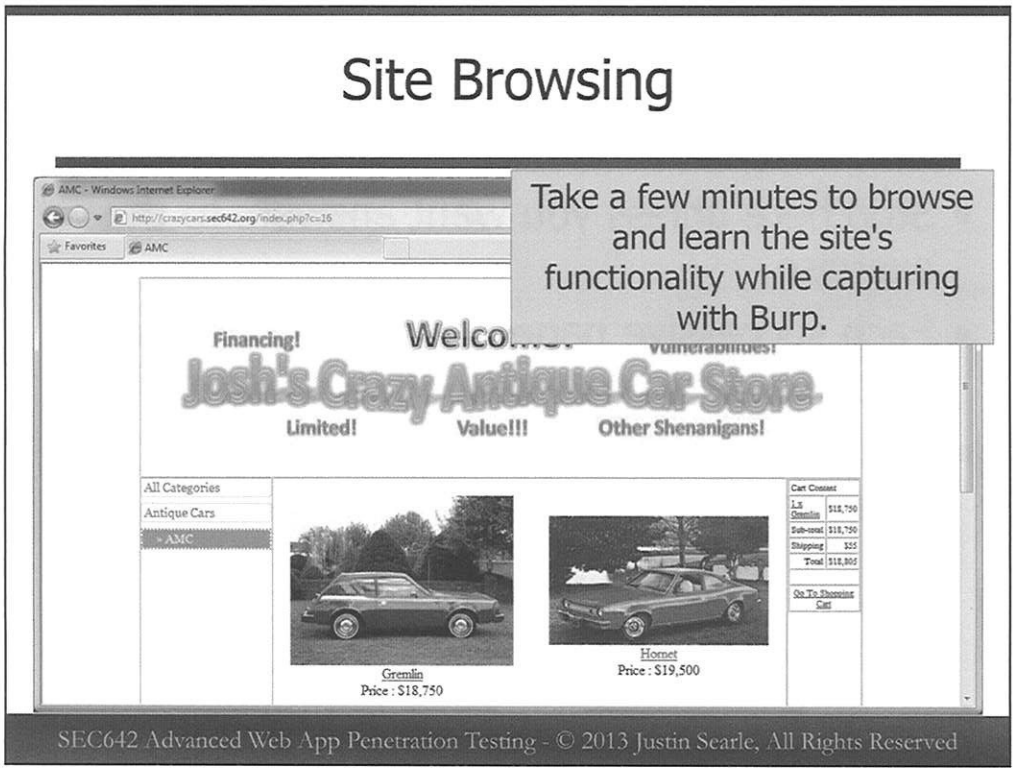
Due to the character analyses change in Burp 1.5, please use the Burp 1.4 to do this exercise. You can use this older version of Burp by typing the following commands:

```
$ cd /opt/samurai/burpsuite
$ java -jar burpsuite_v1.4.01.jar
```



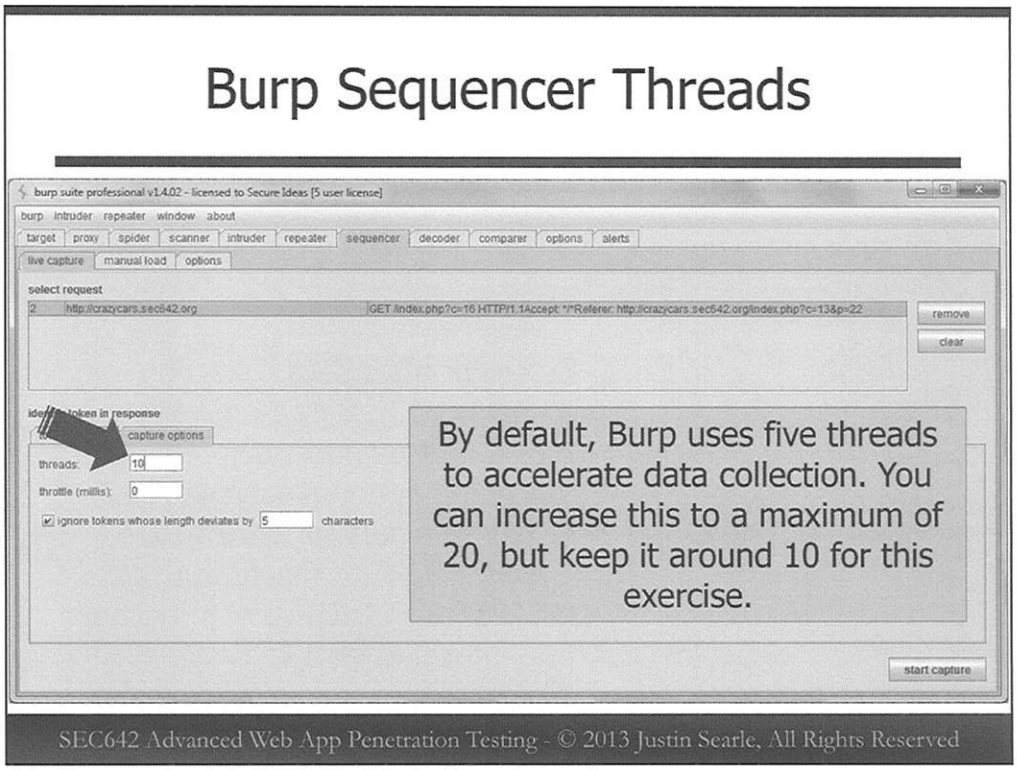
Entropy Analysis – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



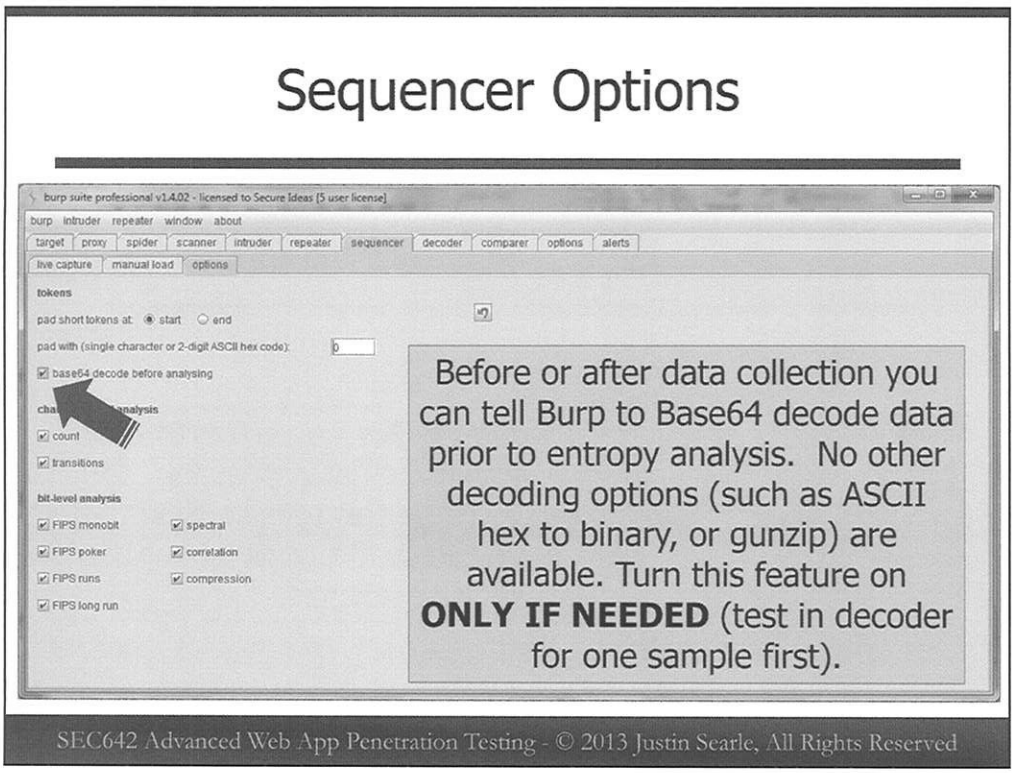
Site Browsing

The Crazy Cars site will be a target for a handful of exercises. Take a few minutes to familiarize yourself with the site while capturing the transaction with Burp. If your browser is configured with NoScript, it will be necessary to turn off this functionality, or add the crazycars site to an exception list.



Burp Sequencer Threads

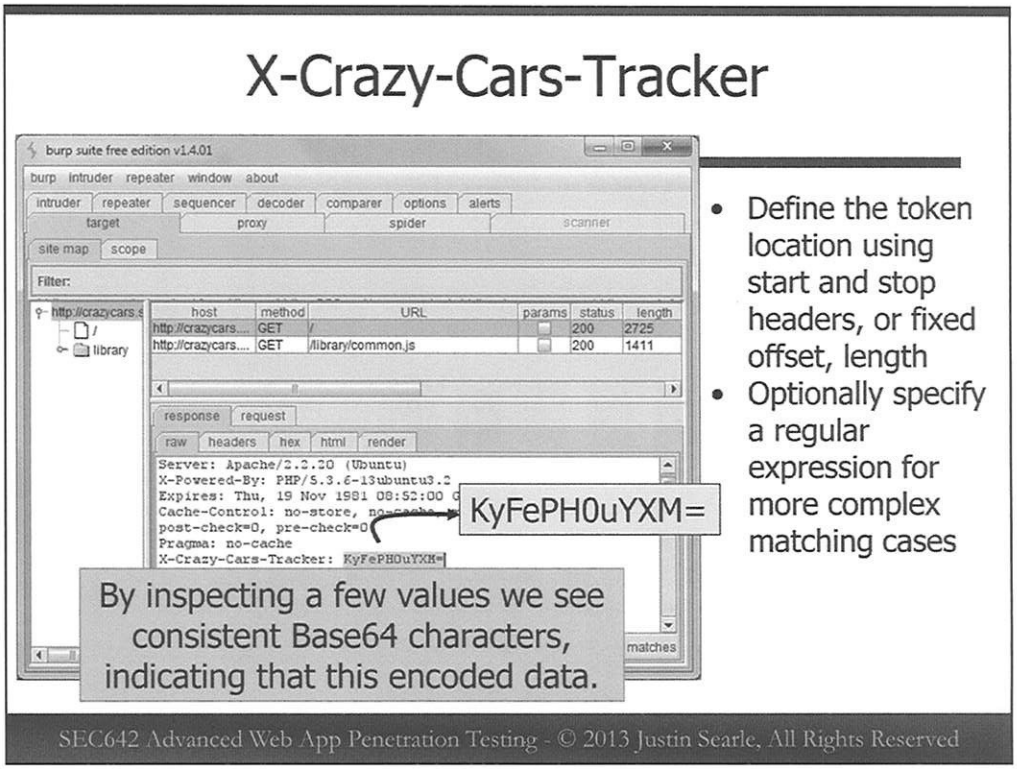
When using Burp Sequencer, we can change the default number of simultaneous threads to a maximum of 20 from the default of 5 threads. For this exercise however, please keep the maximum thread count to no more than 10.



Sequencer Options

It is important to configure Burp to decode the collected data prior to be analyzed for randomness qualities. For Base64 encoded data, navigate to the Options tab and select the "base64 decode before analysing" option. Prior to turning on this feature, ensure the data you are analyzing requires using Burp Decoder.

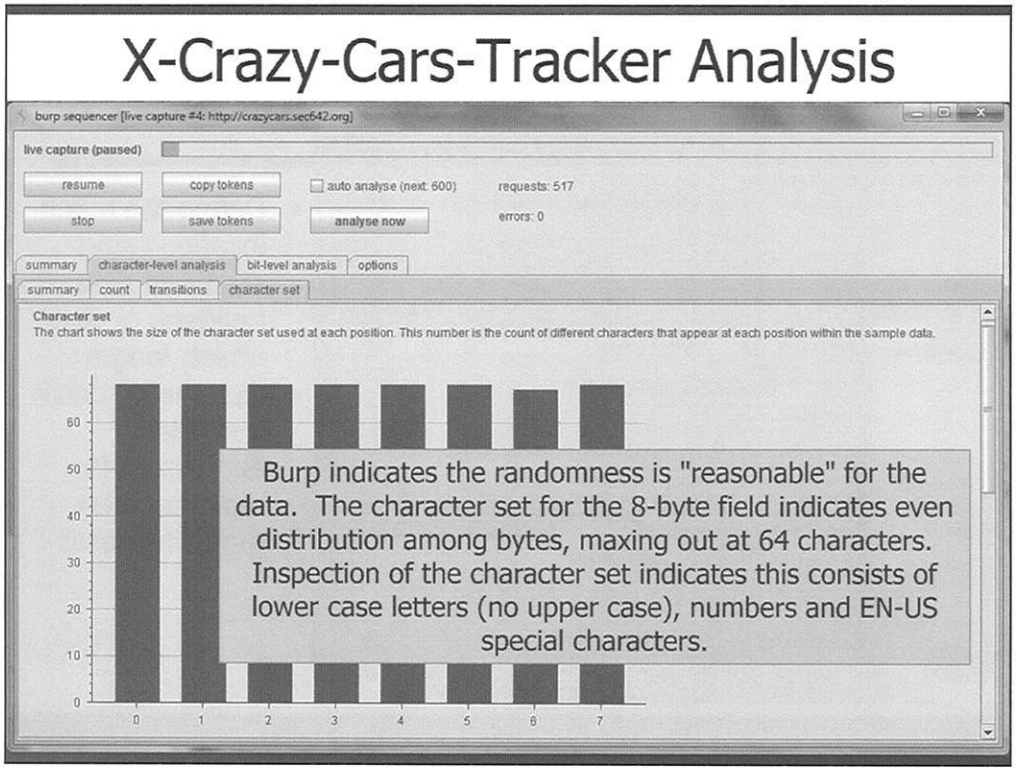
Note that Burp is limited in that you are unable to do any decoding of content for Sequencer other than Base64. If you were to sample data that was hex or URL encoded, for example, you would need to save the samples to a file, write a script to convert the data to its natural form, then reload the data into Burp for analysis.



X-Crazy-Cars-Tracker

The X-Crazy-Cars-Tracker header can be observed in each response for the web server. Looking at the field we can determine that it is Base64 encoded due to the presence of characters common to Base64 encoding including uppercase and lowercase letters, numbers, "/", "+" and the trailing equal sign as shown on this slide.

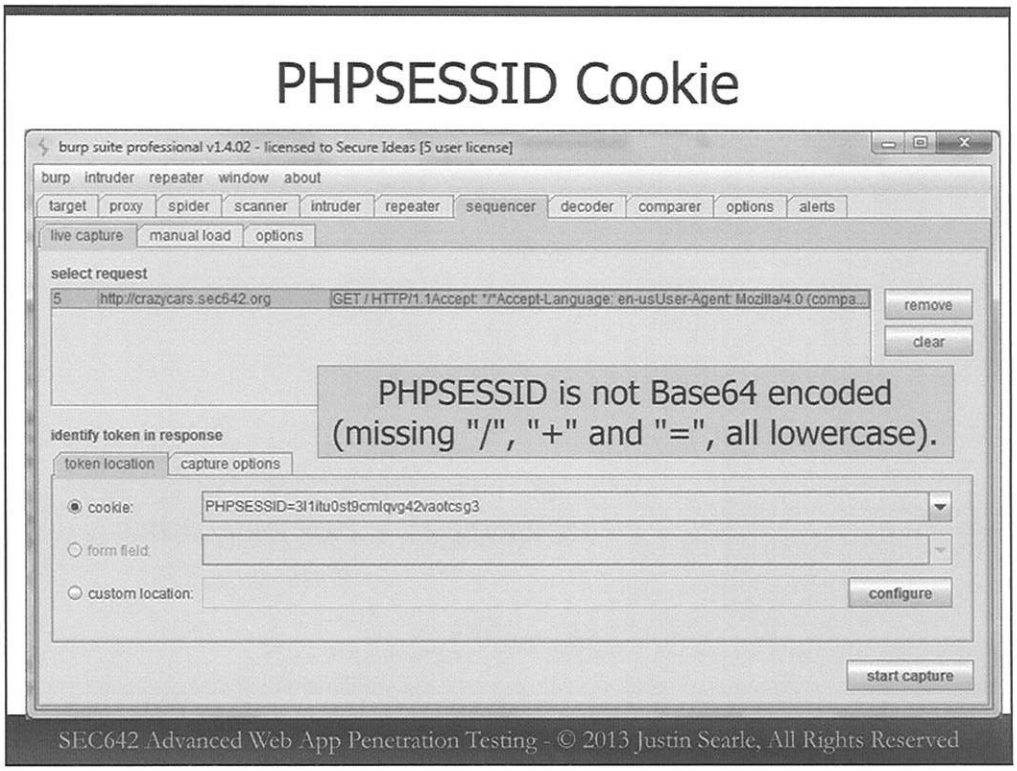
After sending the response data to Sequencer, click the options tab and select "base64 decode before analysing". Return to the live capture tab and highlight the value following the X-Crazy-Cars-Tracker header, then click "start capture". This will launch the Burp Sequencer live capture window, showing you the number of samples collected. After approximately 1,000 data samples, click "stop", then click "analyse now."



X-Crazy-Cars-Tracker Analysis

Burp will likely indicate that the randomness of the data is "reasonable" for the collected data set. The limited size of the field (8-bytes) causes Burp to limit its assessment of the data as having reasonable entropy, despite the high entropy of the collected data.

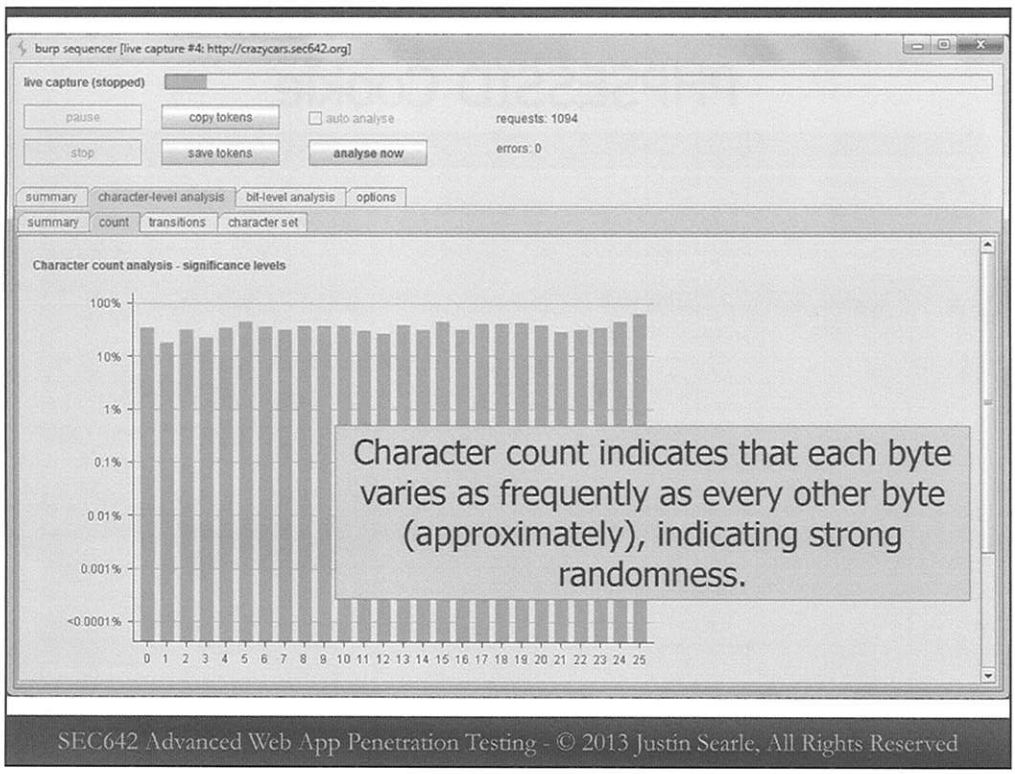
For this data, it is difficult to determine if it is encrypted or not. The data is 8-byte which could be one DES block, but it could also simply be a randomly selected value used for tracking purposes.



PHPSESSID Cookie

The web server will also set the PHPSESSID cookie when the session starts to use it for state tracking. Examining the content of this cookie we see a combination of lowercase character and numbers, but no uppercase characters and nor any "/" or "+" characters commonly seen in Base64 encoding. For this reason it is unlikely that the data is Base64 encoded. Before analyzing the data, ensure the "base64 decode before analysing" option is disabled.

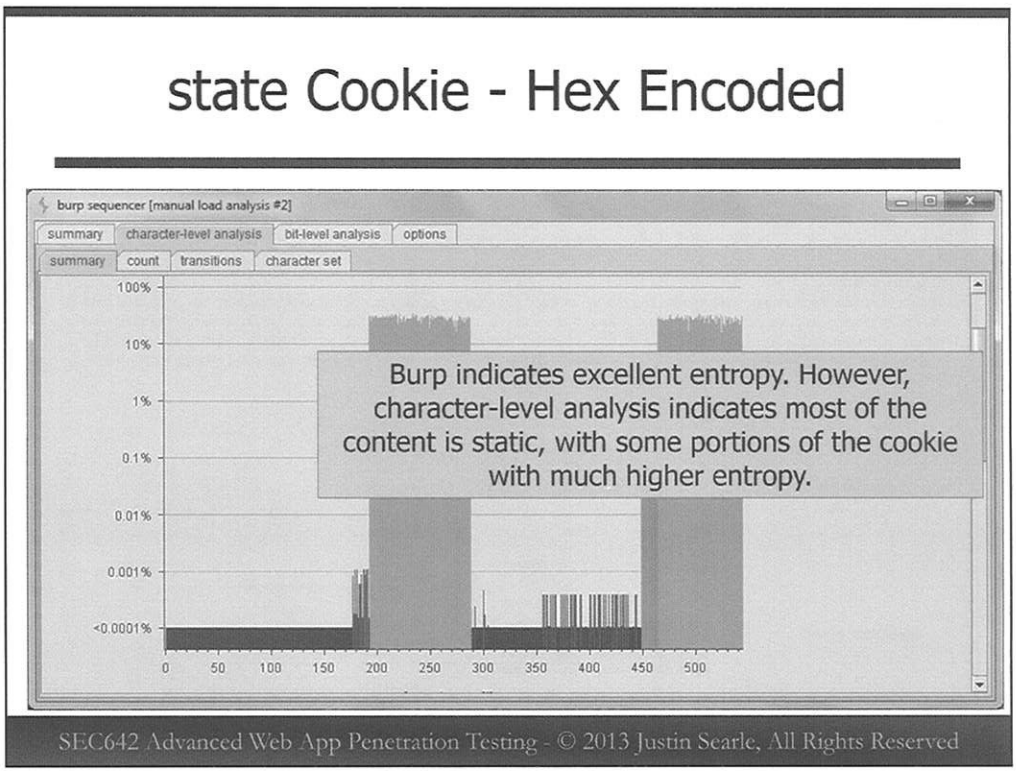
If you do not see a response that sets the PHPSESSID cookie, clear the browser history and restart your browser before returning to the site. The Crazy Cars site will only set the cookie when the session starts.



PHPSESSID Analysis

Burp indicates that the PHPSESSID cookie content randomness is excellent. Examining the character level analysis count tab we can see that, for each of the 25 bytes, the distribution of characters is approximately equal, a strong indicator of randomness.

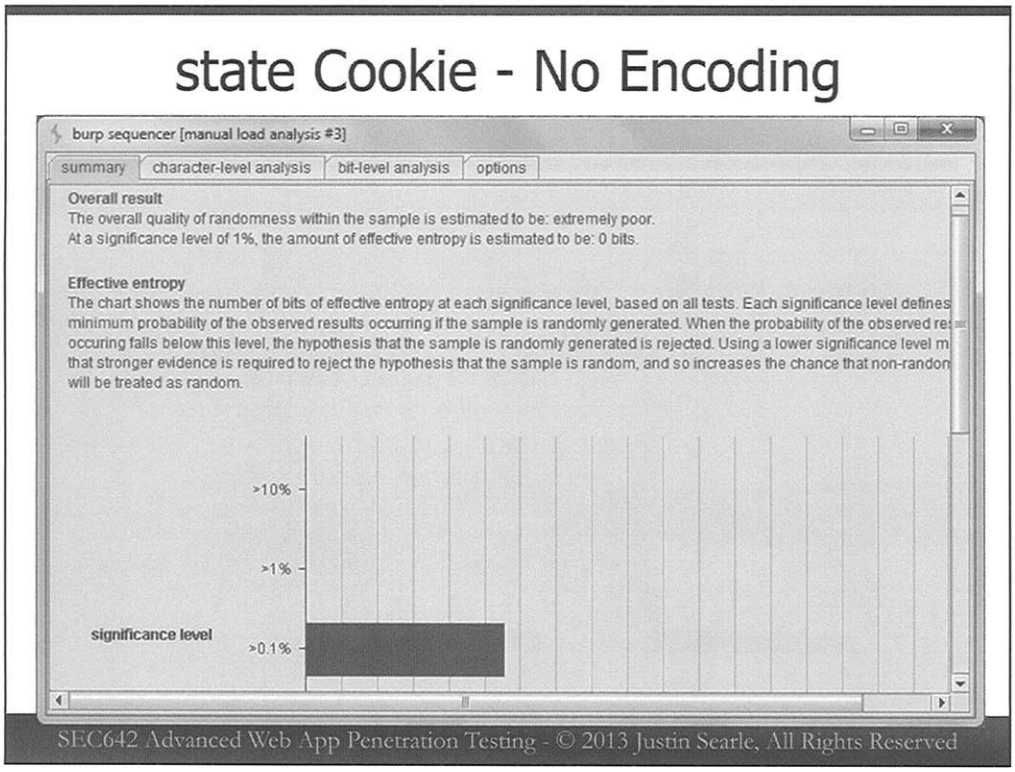
The PHPSESSID value is not encrypted, due largely to the limited character set used for the generation, as well as the fact that the content is generated with a MD5 hash (or other hash value, as specified in the php.ini file) prior to being encoded with 5-bits per byte.



state Cookie

The "state" cookie has interesting properties. It makes an appearance after adding an item to your cart, growing in length as you add more items but always a length evenly divisible by 16. Looking at the encoding of the value, we see strings similar to "8f658bfb0e48ca334cc1c75807ff9c", indicating that the cookie content is being set with hex encoding. Since the ASCII string grows in 16-byte blocks, and each byte of the string indicates half a byte of unencoded data, we know that the block size of 8 bytes.

In the Burp analysis of the cookie content we see a randomness rating of excellent, however, viewing the character level analysis tab we see that only portions of the cookie change with any consistency (noted in green).



state Cookie - No Encoding

Unfortunately, Burp does not have the native ability to return the hex-encoded values to their unencoded form for the Sequencer analysis. We can use Burp to evaluate the content as-is as shown in the previous slide, or we can export the tokens to a file with the "save tokens" option, then decode the tokens and save the natural values to a new file for processing with a short Python script shown below:

```
#!/usr/bin/env python
import sys
import binascii
import base64

fin = open(sys.argv[1], "r")
fout = open(sys.argv[2], "wb")
i=0
for line in fin:
            i+=1
            out=binascii.unhexlify(line[:-2])
            fout.write(base64.b64encode(out)+"\n")
            print "Processed line %d (len %d)"%(i,len(out))

fin.close()
fout.close()
```

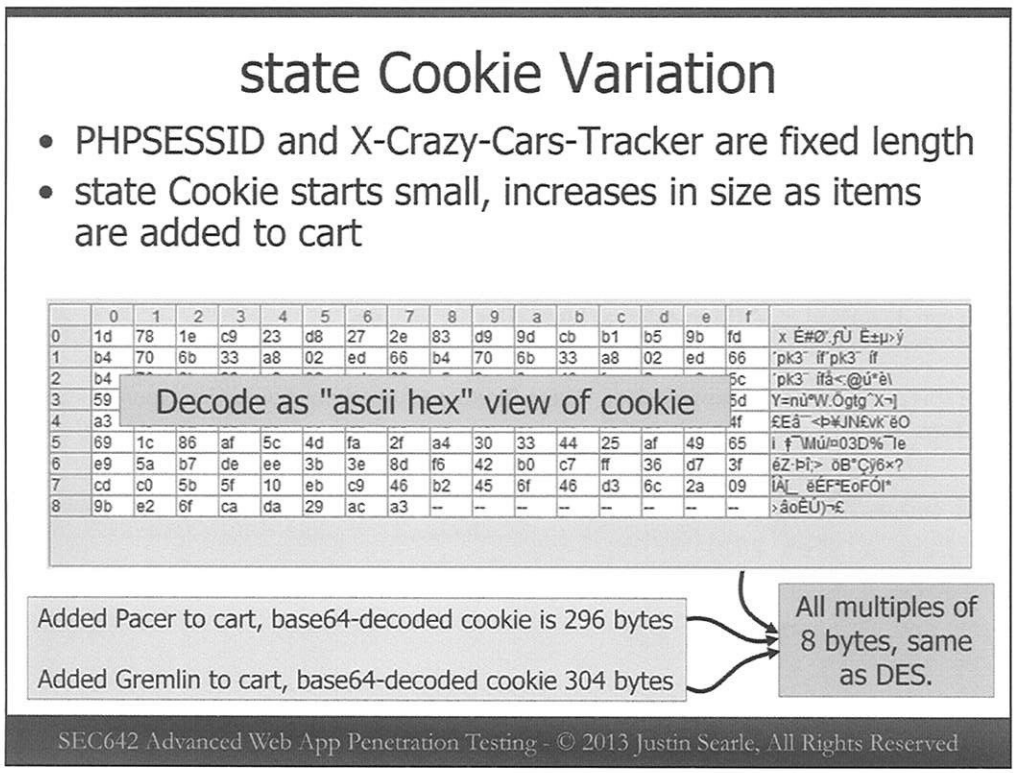
Note that the script converts from hex-ASCII format to natural bytes before writing the value out as Base64. Burp relies on a carriage-return character to denote the end of each record. Saving the records in their natural binary format will cause Burp to misinterpret the end of the token each time a carriage-return character is encountered, even if it is part of the cookie itself and not an end of record marker. To resolve this, we save the data in Base64 format with a carriage-return at the end of each record, leveraging Burp's Base64 decoding functionality when processing the records.

This script can be retrieved from http://files.sec642.org/unhexfile.py.txt. You may run this script on any platform with a Python interpreter, using Samurai or your native operating system. If you need to install Python for Windows, you can download the MSI installer at http://files.sec642.org.

An example of this script in use is shown below, where "state-cookie-tokens.txt" is the exported state cookie token content from Burp:

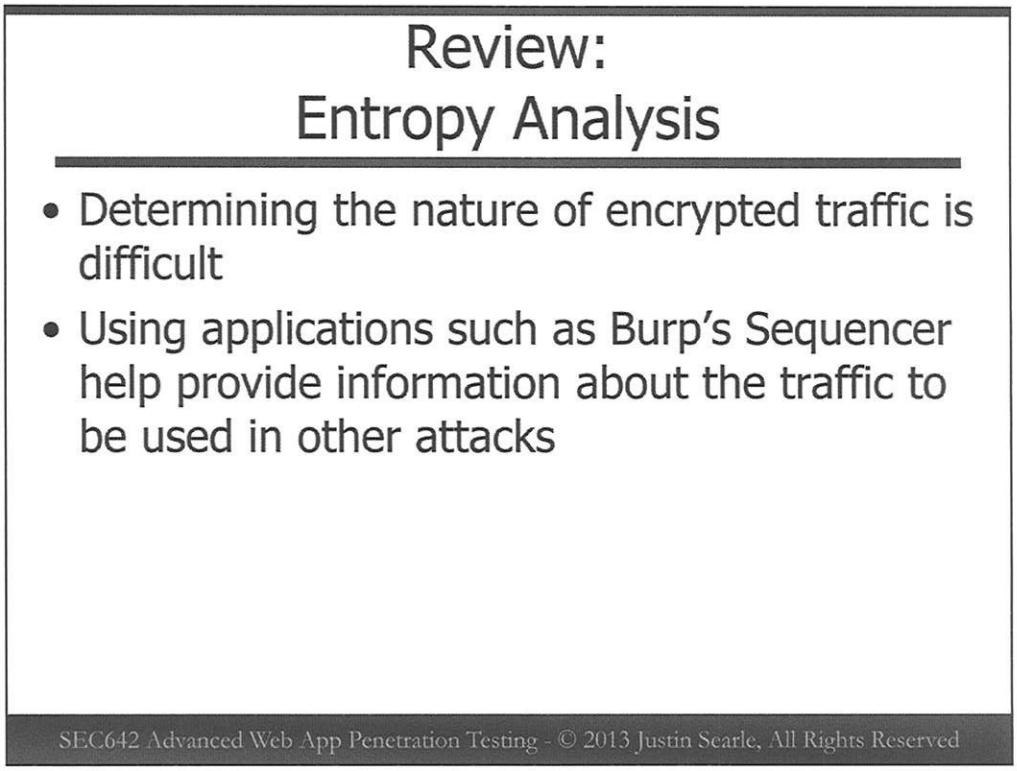
```
C:\>python unhexfile.py state-cookie-tokens.txt state-cookie-tokens.bin
Processed line 1 (len 272)
Processed line 2 (len 272)
Processed line 3 (len 272)
...
```

The length of the object after decoding from hex-ASCII is shown in the output. Next, return to Burp and select the "manual load" tab in Sequencer. Import the file generated from the script and ensure the option to enable Base64 decoding has been checked on the options tab before selecting "analyse now". Burp's analysis of the cookie content now indicates that the randomness of the content is extremely poor, which is a more accurate reflection of the nature of the data.

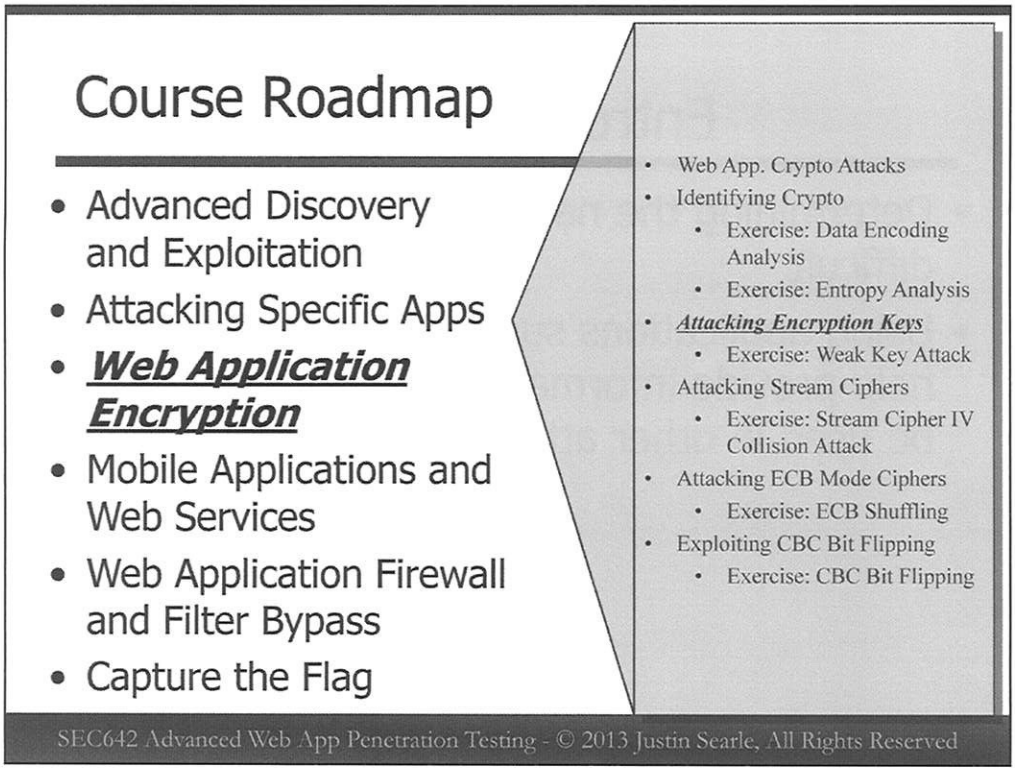


state Cookie Variation

Of the three values reviewed, PHPSESSID, X-Crazy-Cars-Tracker, and the state cookie, only the state cookie begins as a small value and increases in length with 8-byte increments (in the natural byte form). The entropy of the state cookie content is low, and careful examination of the encrypted content reveals numerous duplicate block values, an indicator of ECB encryption. Combined with the use of 8-byte blocks, it is likely that the cookie is encrypted using DES-ECB.

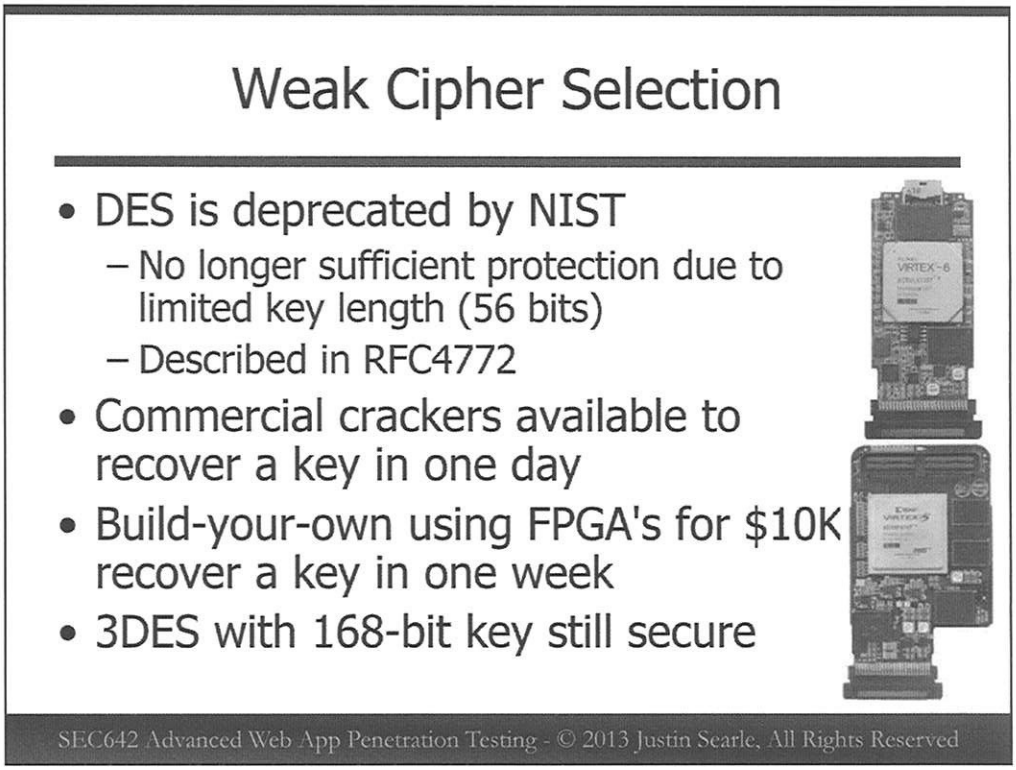


Determining the nature of encrypted traffic is difficult, and there are many cases where we simply do not know how the content is encrypted or hashed. Fortunately, there are still several analysis and attack options available to us, as we'll continue to see in this section.



Course Roadmap

Next we'll look at exploiting web application encryption systems, starting with attacking encryption key selection.



Weak Cipher Selection

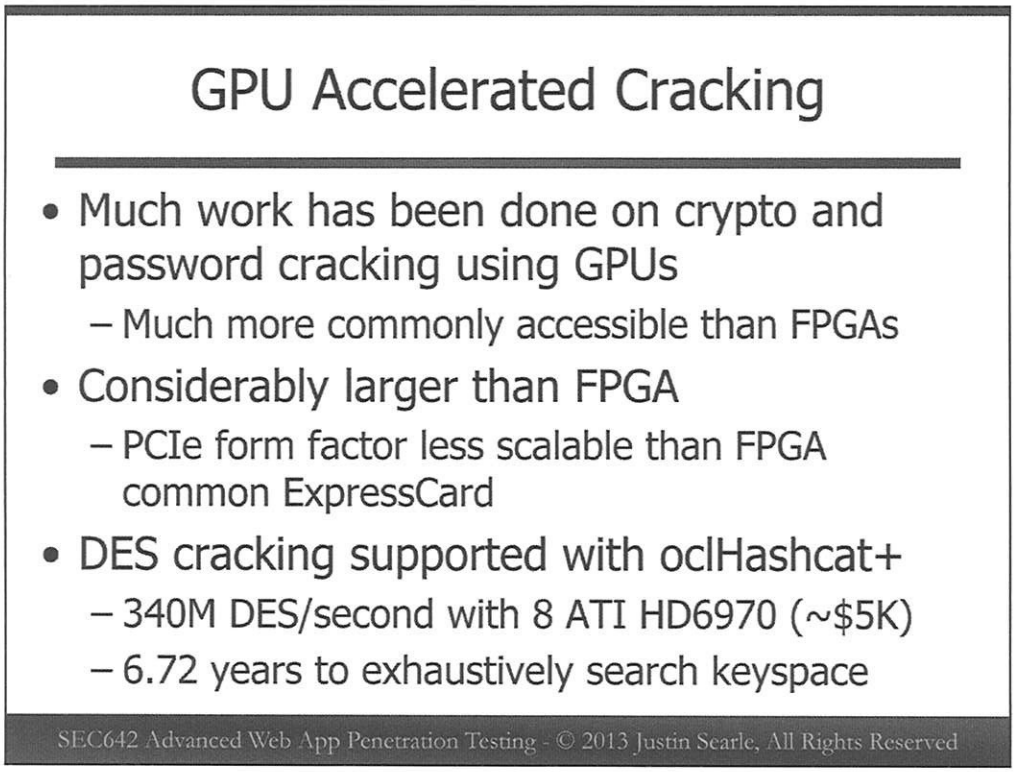
Traditionally, many penetration testers focus on algorithmic weaknesses in ciphers for use as an attack opportunity, targeting opportunities such as an insufficient key length or a weak cipher. In some cases, this is a practical attack, such as attacking systems using the legacy DES cipher.

DES is no longer a NIST-recommended cipher due to weaknesses in the limited key length of DES (DES requires 56-bit keys). Additional details on the weaknesses and risks of using DES are further documented in RFC4772.

Despite DES being a weak cipher due to key length limitations, it is not a trivial task to recover an unknown DES key. Commercial DES cracking services from companies such as COPACOBANA (http://www.copacobana.org/) or commercial products such as the Tableau TACC1441 (http://www.tableau.com/index.php?pageid=products&model=TACC1441) are available to attack and recover DES keys.

Alternatively, for approximately $10,000 USD it is possible to build your own DES cracker with Field Programmable Gate Arrays (FPGAs) which could recover a DES key through exhaustive key space search in one week. One FPGA company, Pico Computing, makes FPGAs with software for DES cracking freely available at http://openciphers.sourceforge.net.

Note that Triple DES, while not recommended by NIST for new applications, is still considered secure as there are no practical attack opportunities available.



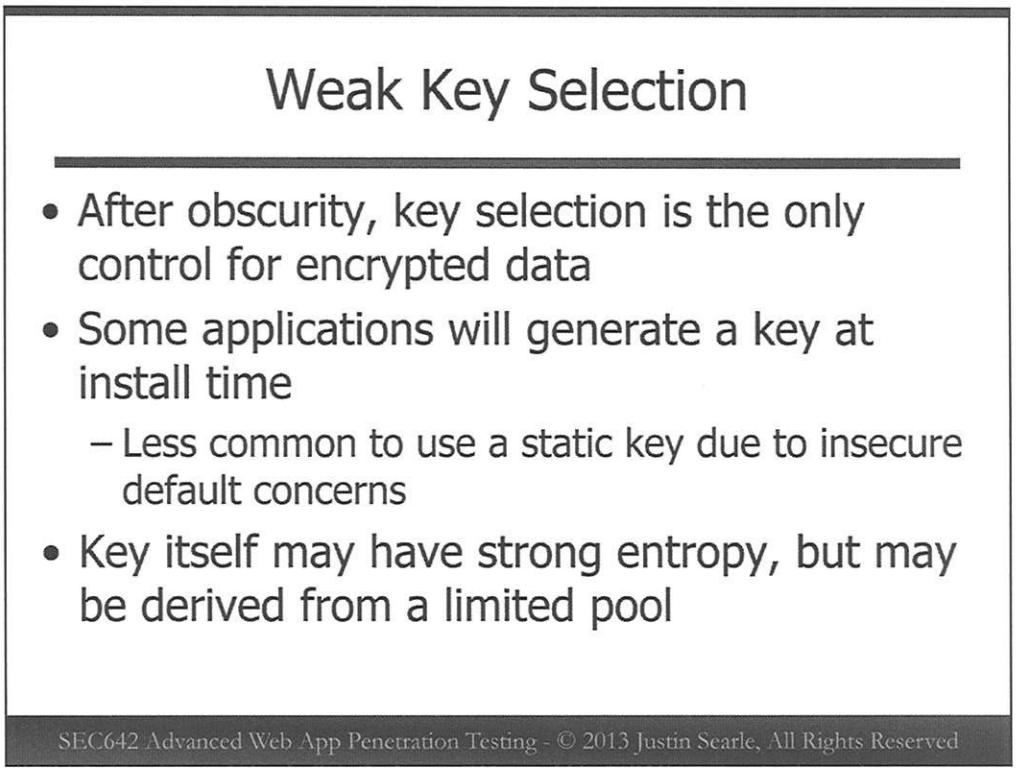
GPU Accelerated Cracking

In the past several years, the general availability of high-power Graphics Processing Units (GPUs) has led to development efforts for cryptographic acceleration to improve greatly over what has been possible on traditional CPUs. This is largely due to the relatively low-cost and widespread accessibility of GPUs, compared to limited availability of FPGAs.

While GPUs are more widely available than FPGAs, they do not perform at the same level FPGAs, and with a considerably larger form factor, offer less scalability than systems with PCI-Express card slots for multiple GPUs.

Leveraging GPUs for DES cracking support, the oclHashcat+ project (http://hashcat.net/oclhashcat-plus/) achieves a rate of 340 million DES operations a second distributed across 8 ATI GPUs. A comparable system can be built for approximately $5,000 USD. At a rate of 340 million DES operations a second, it would take 6.72 years to exhaust the DES key search space.

As a web application pen tester, we should keep in mind that DES is a weak cipher due to its limited key length, and that an adversary with sufficient funding or resources (such as a botnet) could recover the key through a brute-force effort. However, there are other attack opportunities present in some systems stemming from weak key selection vulnerabilities.

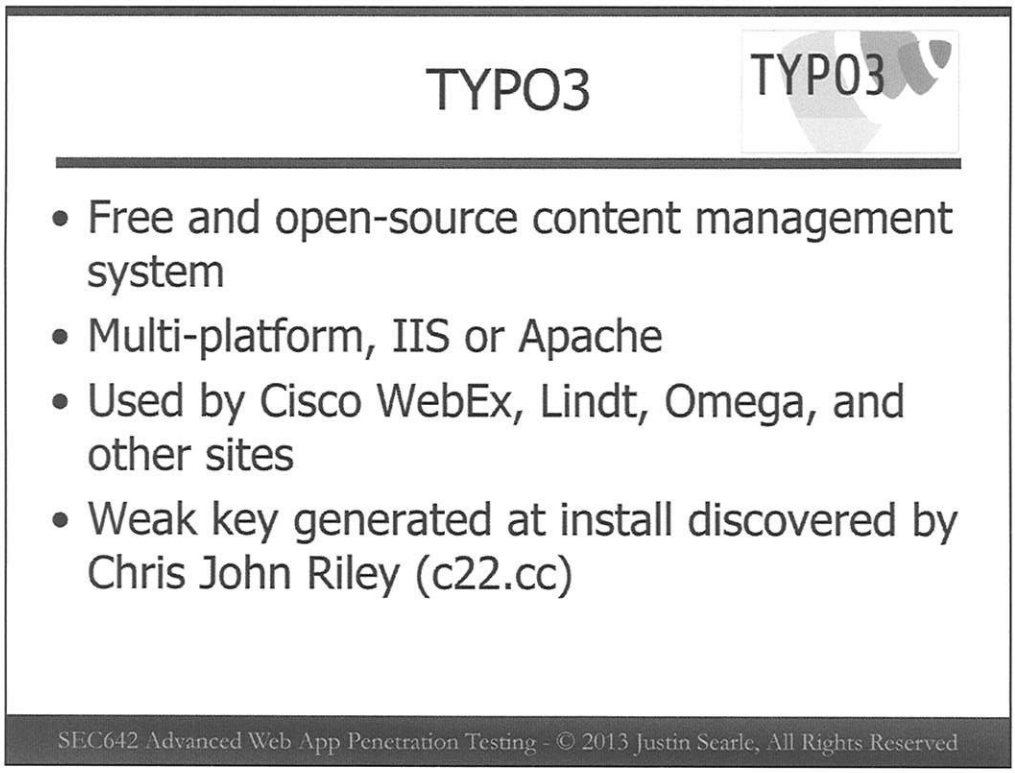


Weak Key Selection

The selection of the cryptographic key used to protect the system is a vital component. While many systems leverage some form of obscurity for additional security, these controls can always be overcome, leaving the encryption key the only remaining unknown barrier to the attacker.

Historically, users make poor choices when selecting sensitive content such as encryption keys. For this reason, many commercial and open-source software products have turned to generating keys automatically at install time. The process of selecting the key should be carefully scrutinized.

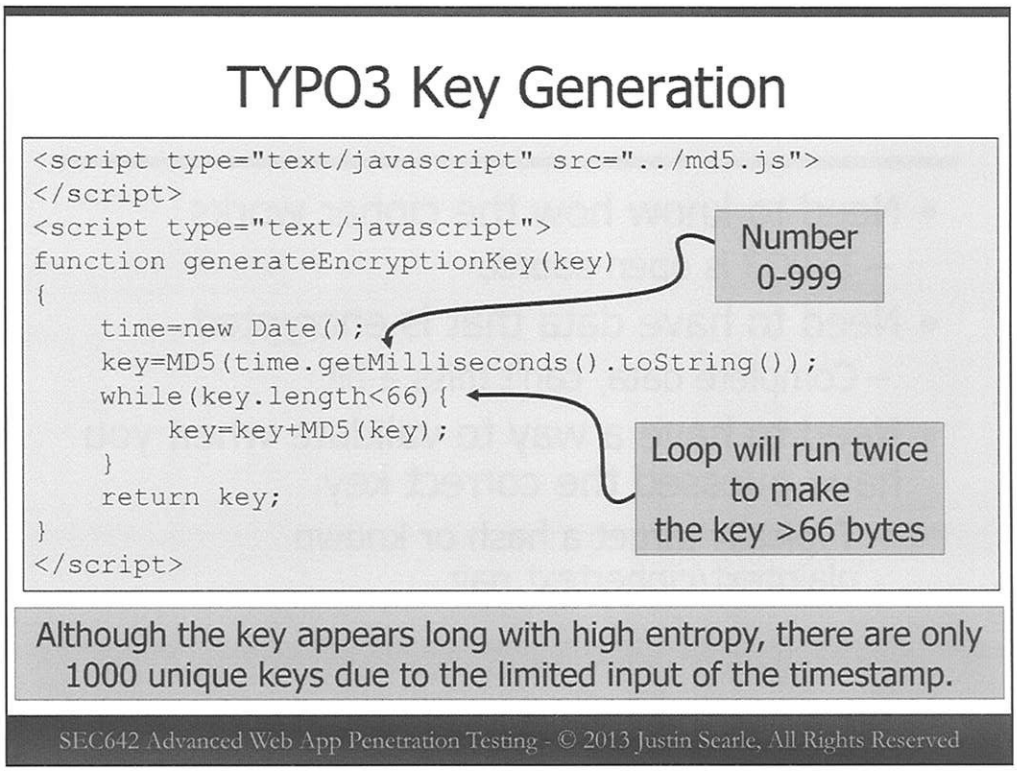
In many cases, a low-entropy input value is used as part of an MD5 or SHA1 hash, producing a key value that appears to have high entropy but is actually weak. We'll examine one such case in the default key selection vulnerability of the TYPO 3 CMS.



TYPO3

TYPO3 is a free and open-source content management system for Windows or Unix hosts, leveraging IIS or Apache. On their website, TYPO3 claims many high-profile sites as being powered by their technology including Cisco WebEx, Lindt Chocolates, Omega Watches and other sites.

TYPO3 installations using a default encryption key were exposed to a weak key selection weakness discovered and reported by Chris John Riley of www.c22.cc.

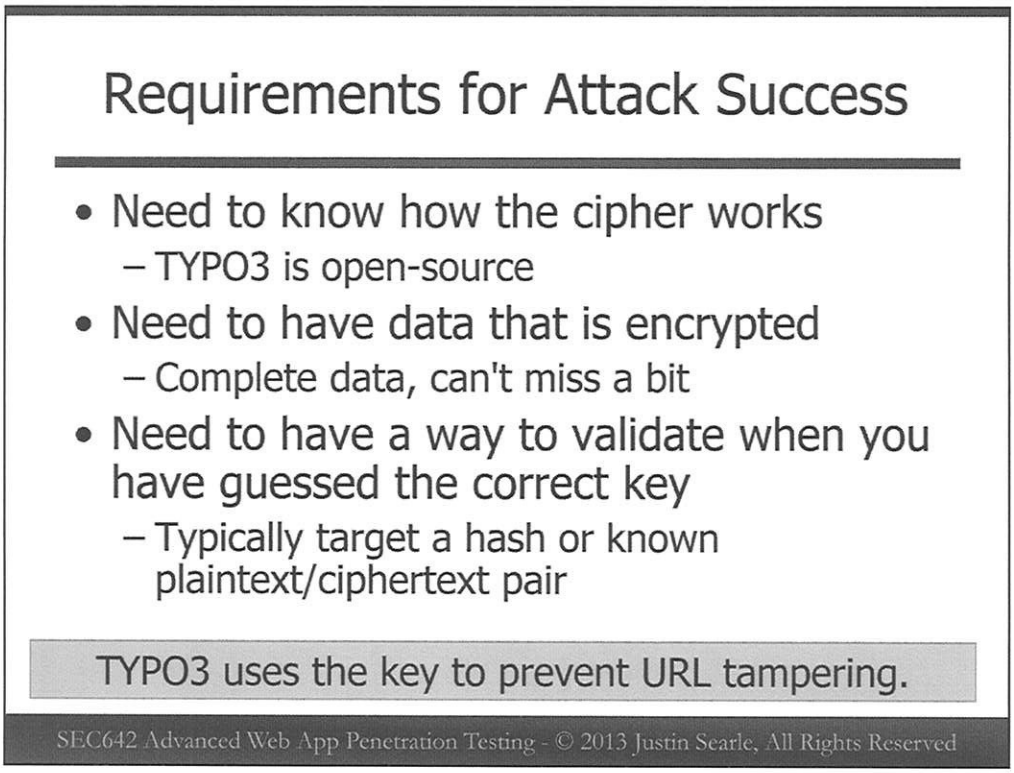


TYPO3 Key Generation

This slide shows a code excerpt from the TYPO3 software used to generate the default encryption key. First, the page loads a JavaScript file "md5.js", presumably implementing the MD5 hash function in JavaScript. Next, the generateEncryptionKey function is shown, creating a variable "time" from the Date() method. The variable key is set to the MD5 hash of the current timestamp millisecond counter, converted to a string value.

Since there are only 1000 milliseconds per second, the value used as the input to the MD5 hash is within the range of values between 0 and 999. The key value itself is the result of several hashes to extend the key length to 66 bytes or more, but the input entropy is extremely limited, creating a weak key exposure vulnerability.

TYPO3 corrected this flaw in version 4.2.4 with a key selection mechanism that uses higher entropy. However, earlier installations where a vulnerable key was selected remain vulnerable even after the update is applied unless the administrator generates or enters a new key manually. For this reason, even current TYPO3 installations may be vulnerable to a key recovery attack.

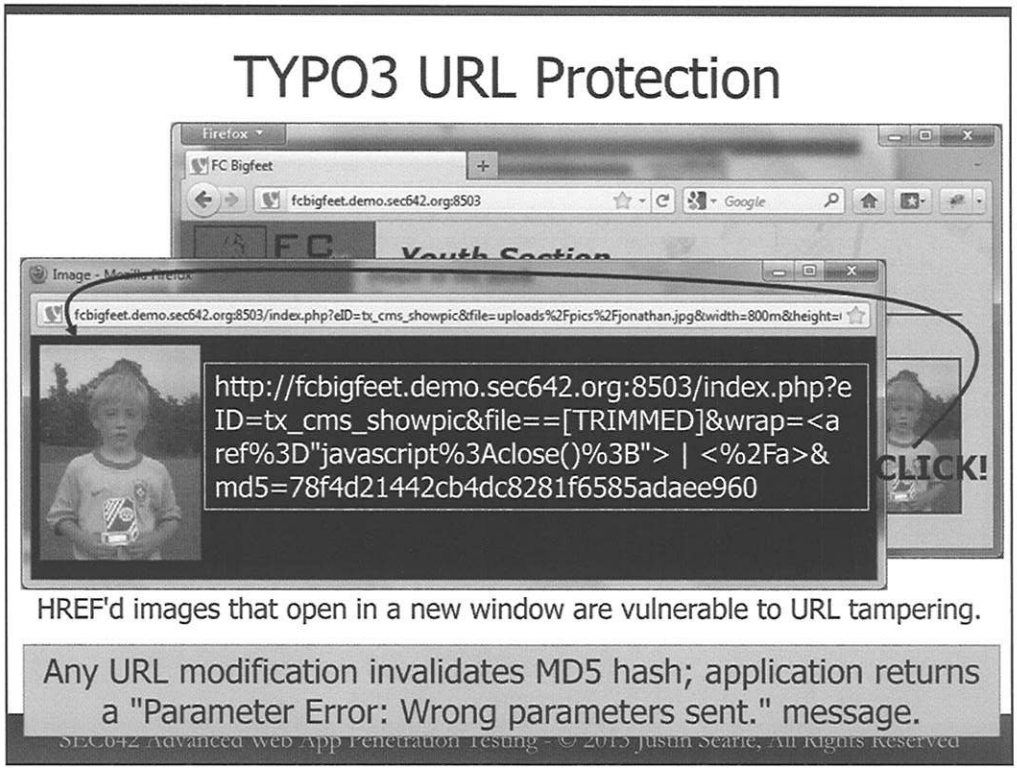


Requirements for Attack Success

In order to exploit a weak key selection mechanism, we need to meet several requirements:

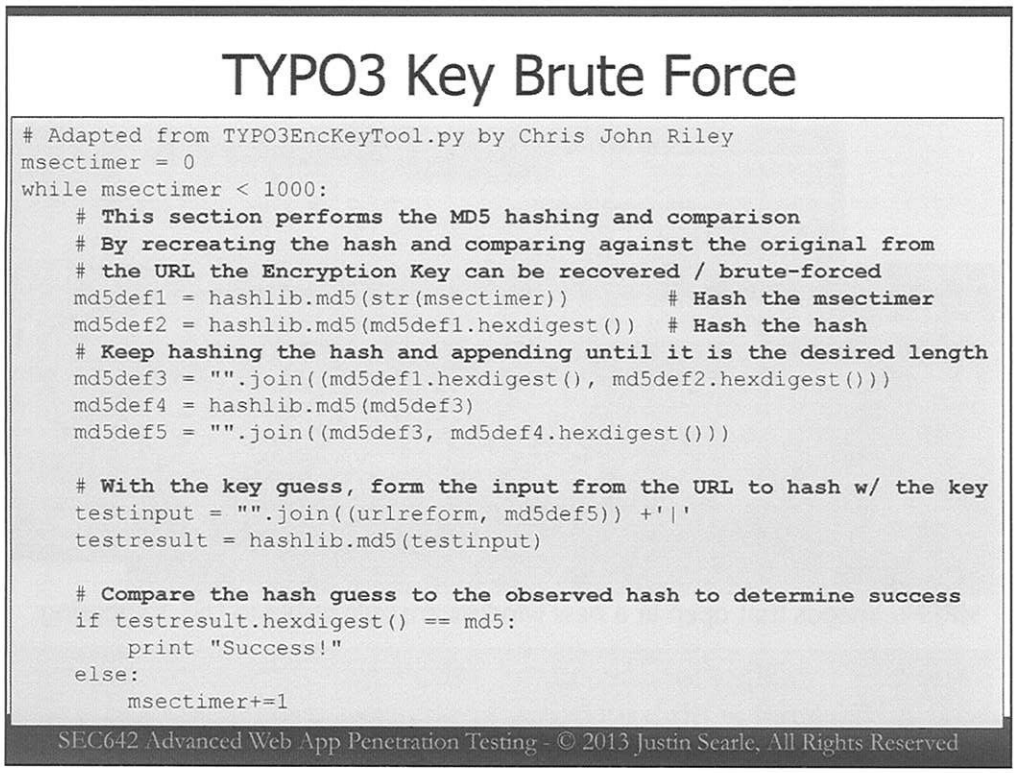
- We need to understand exactly how the cipher works, including what is included in encrypted or hashed content, and how they key is used.
- We need to have a complete encrypted or hashed data set. We need to have all the input that goes into the encryption or hashing process to produce ciphertext.
- We need a way to validate when you have guessed the correct key. When guessing keys, we need to identify a mechanism that we can use to identify the correct key guess. Since we can't compare our key guess to anything, this is typically done by knowing the input plaintext and encrypting the content with our key guess, comparing our encrypted results to the observed encrypted results, similar to what is done in an authentication protocol.

In the case of TYPO3 we meet all these requirements since TYPO3 is open-source. TYPO3 uses the encryption key to prevent URL tampering attacks, leveraging a portion of the URL to generate a hash that is later appended to the URL, giving us the complete input data set and a hash to compare our guess key encryption result to.



TYPO3 URL Protection

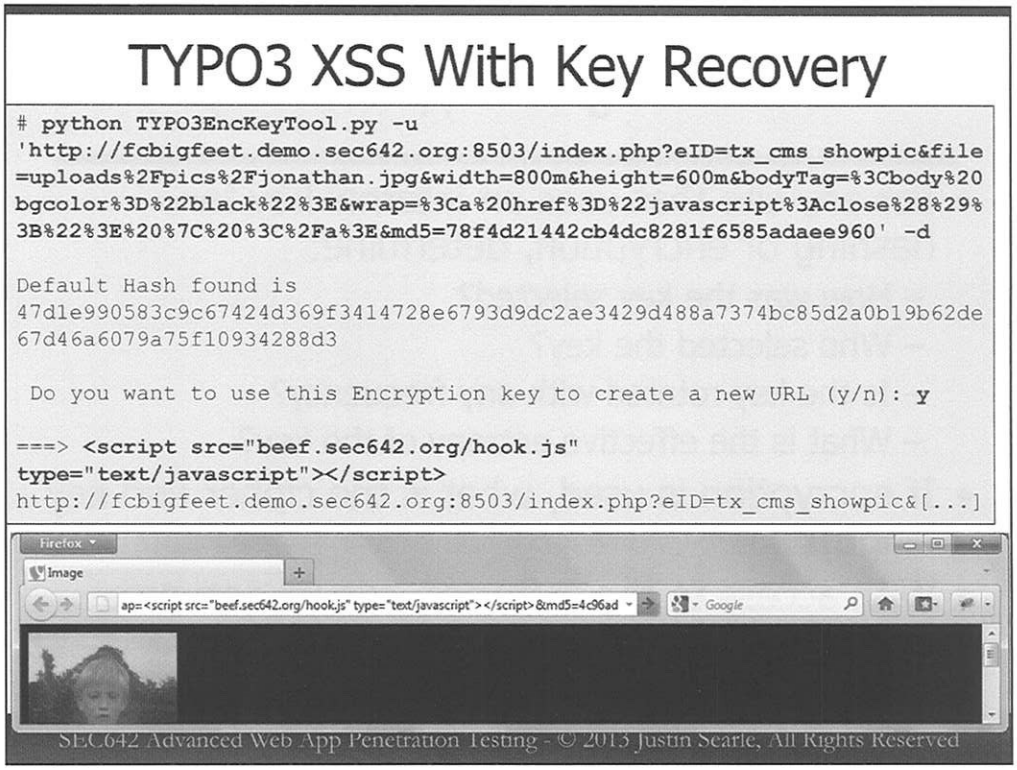
TYPO3 uses the weak key as part of a mechanism to protect against URL tampering for HREF'd images. When you click on a HREF'd image in TYPO3 it will open in a new window as shown on this slide. The image URL included multiple GET parameter settings for the display of the content, as well as embedded JavaScript to close the image upon clicking it. In order to avoid XSS attacks through URL modification, TYPO3 hashes the URL with the weak key and includes a trailing "md5" parameter in the URL with the hash result.



TYPO3 Key Brute Force

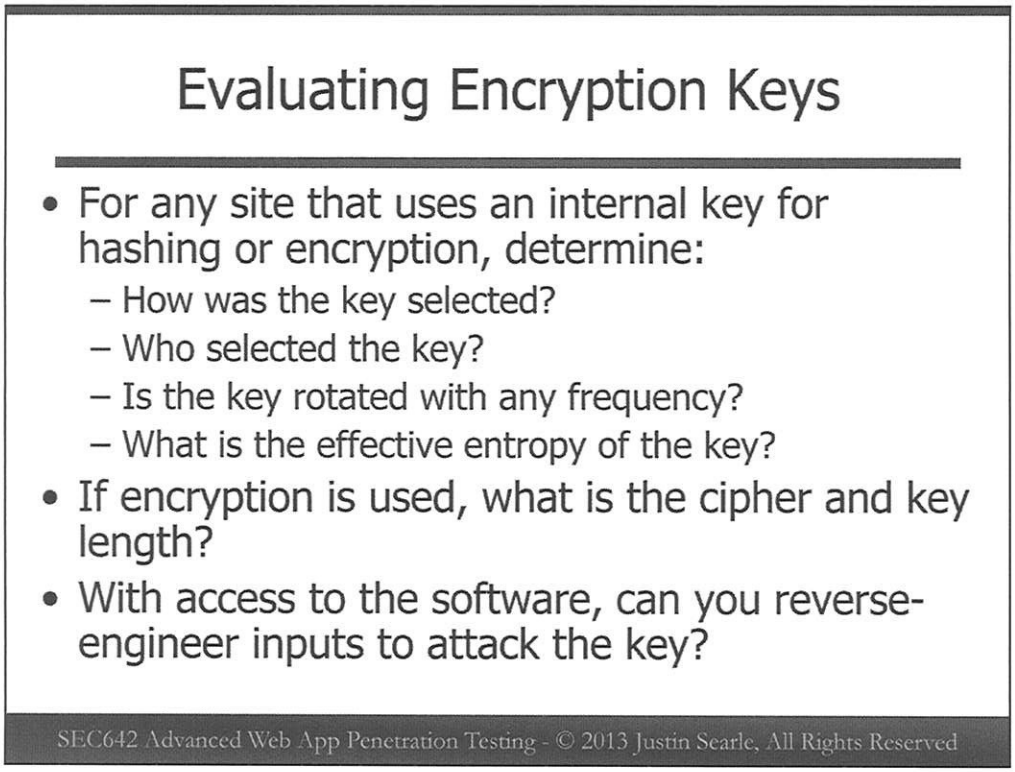
This partial Python script was written by Chris John Riley and adapted with additional comments for SEC642 by this author. This script sets a variable "msectimer" to an initial value of 0 and then performs a loop 1000 times, hashing the msectimer value as a string multiple times in the same fashion that vulnerable TYPO3 installations did. Once the hash is generated, specified URL content is parsed in the same manner that vulnerable TYPO3 versions did and is hashed using the weak key with a vertical bar character separator ("|").

The resulting key and hashed URL content are then compared to the MD5 value present in the URL; if they match, we have found the correct key. If the values do not match, we simply repeat the loop until we find a match or we run out of msectimer values to use as MD5 inputs.



TYPO3 XSS With Key Recovery

This slide shows an example of using the TYPO3EncKeyTool.py script with a target URL ("-u"), performing a default key value search ("-d"). The default hash value is found in short order, due to the limited search space for the msectimer value. With the correct key, we can specify our own JavaScript content to include in the URL. In this example, we add a BeEF hook, leveraging the compromised key for an XSS attack.



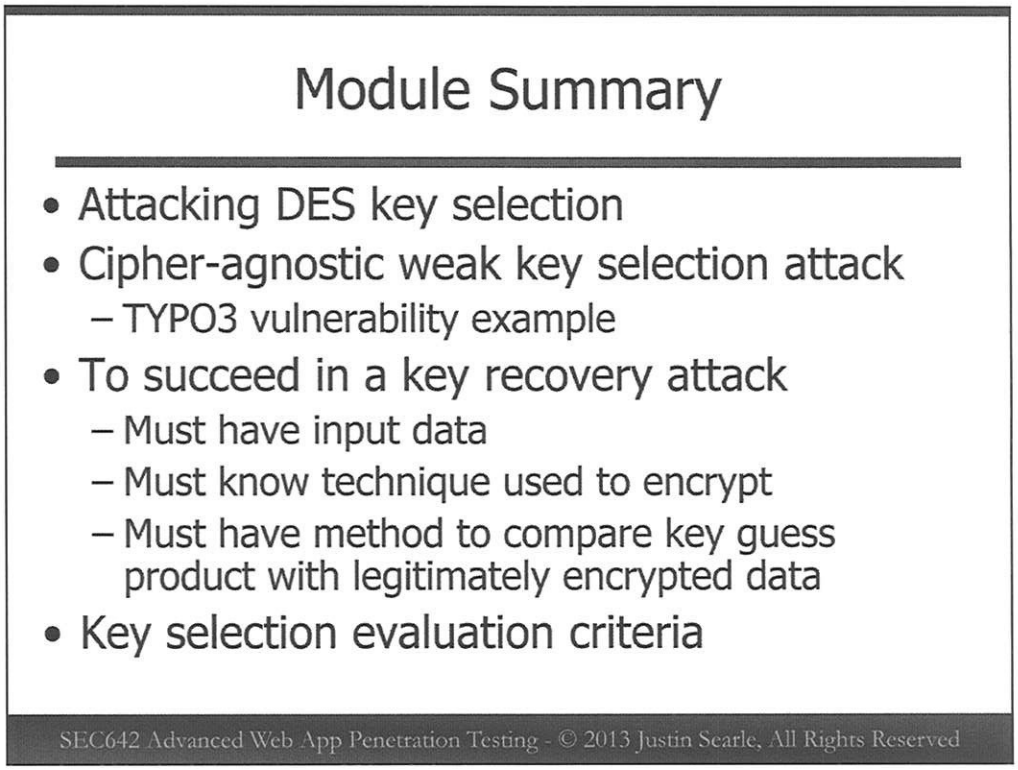
Evaluating Encryption Keys

As part of a cryptographic analysis of any web application, we should examine how keys are selected, including:

- How was the key selected (hashed from a weak value, such as the TYPO3 example?)
- Who selected the key? Was it selected by the administrator, or by the vendor? Could it be a default key that is common across multiple installations that you could recover if you install the software locally?
- Is the key rotated with any frequency or is the key static?
- What is the effective entropy of the key? Remember, input that is hashed with MD5, SHA1 or other hashing functions will appear to have high entropy due to the quality of the hashing function, but if the input value to the hashing function is limited then the effective entropy of the key is very low.

Also evaluate the use of encryption on the system. If encryption is used, what is the algorithm and key length? Can the cipher be exploited in a brute-force attack with sufficient resources? Are the required resources needed to brute-force the key available to the penetration tester , or otherwise available to an adversary the customer is willing to defend against?

In the case of TYPO3, we were targeting an open-source software platform's key selection process. For proprietary software, this vulnerability would have been more difficult to discover since we would not expect to have access to the vulnerable system source code. In the case of proprietary software, we would leverage reverse-engineering techniques to learn how the system encrypts or hashes data and how the key is generated, a much more time-consuming analysis method.

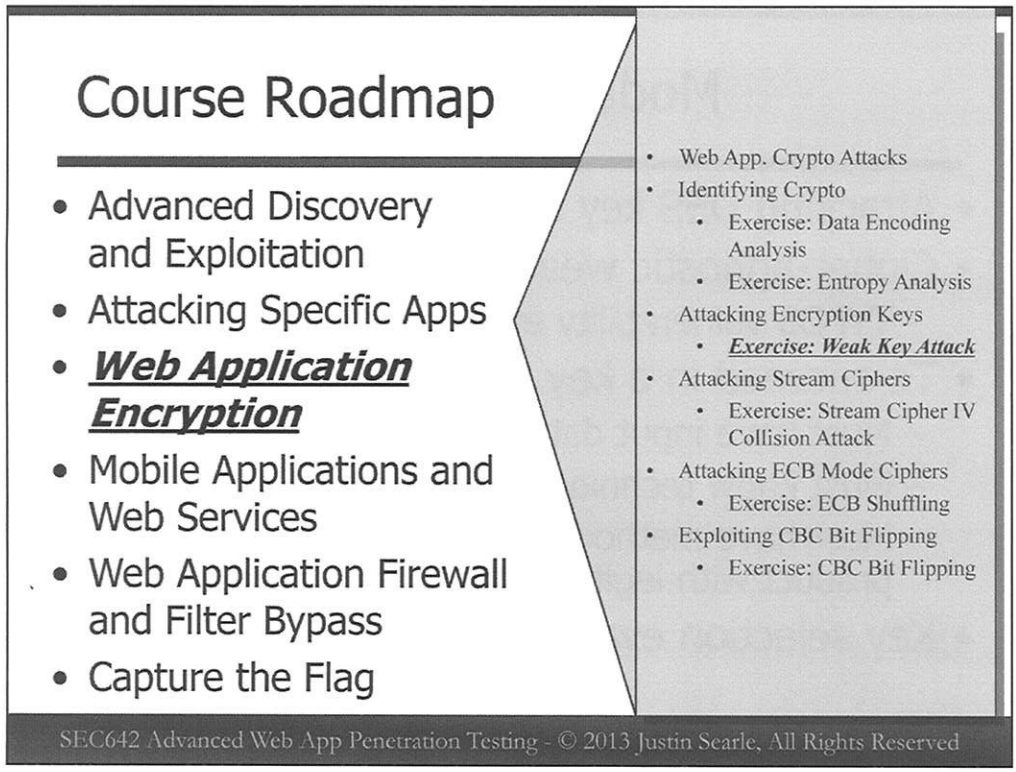


Module Summary

In this module we looked at key selection attacks, brute-forcing weak key length ciphers such as DES through the use of hardware-based cryptographic accelerators and cipher-agnostic key selection weaknesses. We picked on TYPO3 as a common vulnerability example, but the vulnerability there could happen in any other product where a key is generated automatically for the user.

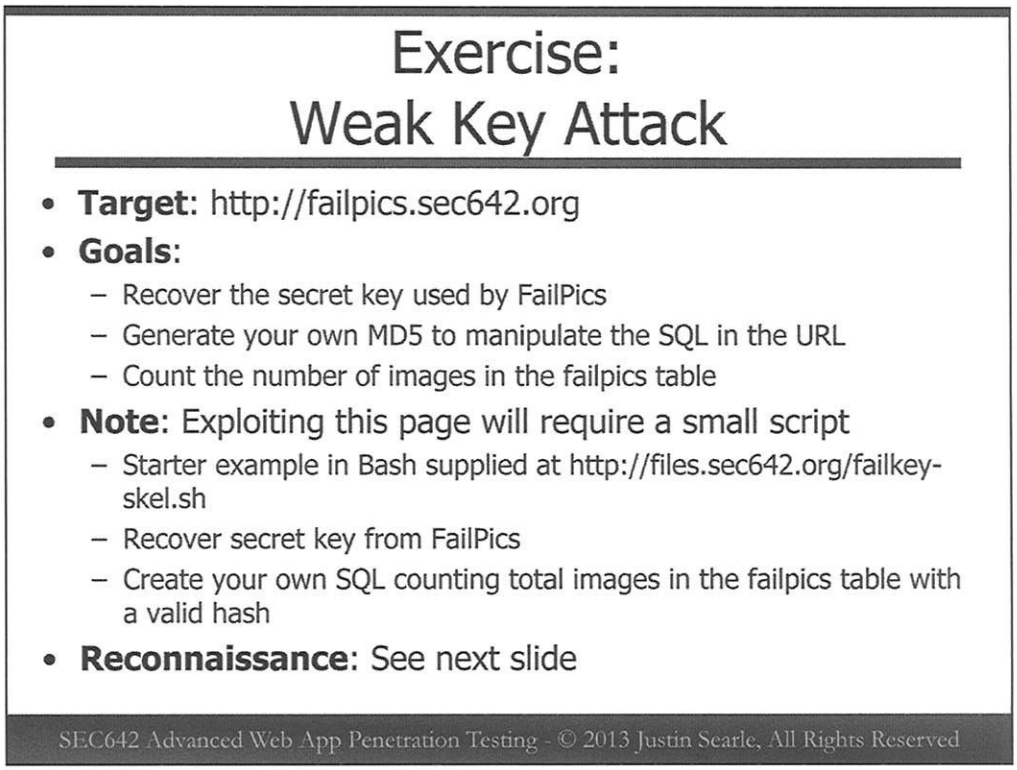
Remember that, to succeed in a key recover attack we must have the input data, the technique used to encrypt or hash the data, and a method to compare our key guess to the legitimately encrypted or hashed data. These requirements can be significant, especially in the case of proprietary software where the inputs to encryption and hashing functions are not known.

Finally, we looked at recommendations to evaluate key selection systems used by web applications. Remember to ask "how was the key selected", "who selected the key", "is the key rotated" and "what is the effective entropy of the key" for any web system using encryption.



Course Roadmap

Now we will do an exercise.



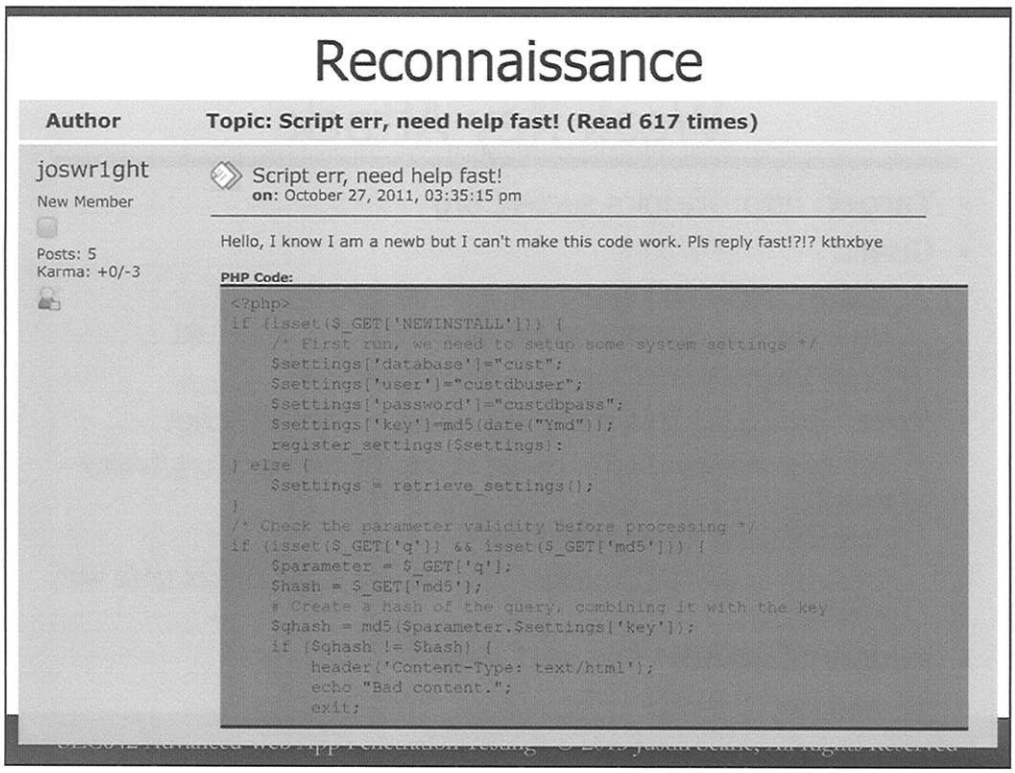
Exercise: Weak Key Attack

In this lab exercise you'll target the weak key selection for the FailPics! site at http://failpics.sec642.org. At the main page, click the "Click for Fail!" button to open a new window that includes a randomly selected image.

Scripting

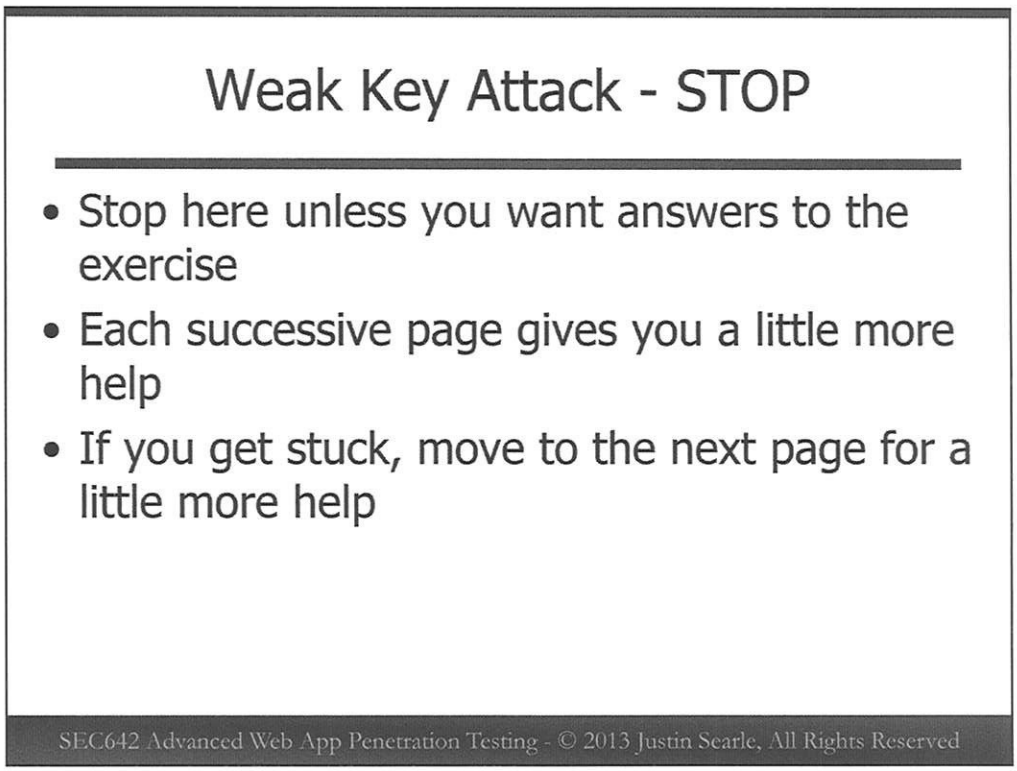
In order to complete this exercise you'll need to develop a small script in a language of your choice. A sample Bash script for the Samurai environment is made available at the URL on this slide. Note that this script is just a skeleton to get you started; you will have to complete this script to use it in your attack.

Your goal is to recover the secret key used by FailPics to manipulate the SQL in the URL to count the number of images in the failpics table. In order to create your own SQL, you will need to produce a value hash using the recovered key.



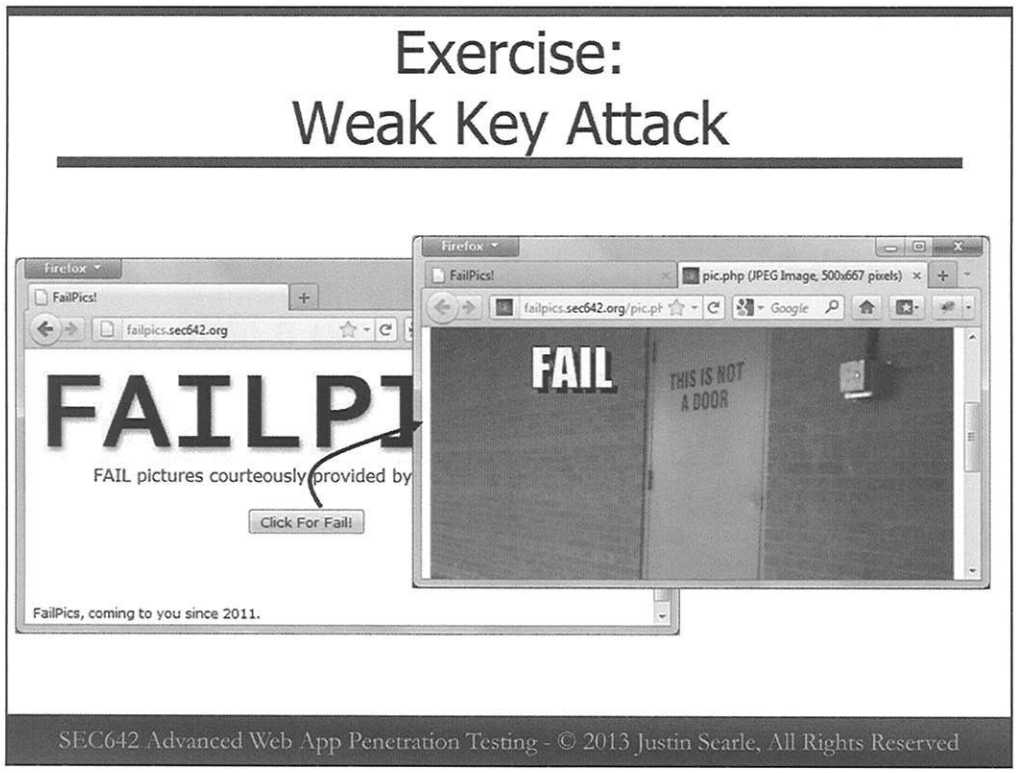
Reconnaissance

During your reconnaissance, you came across the post shown on this page from a public PHP developer's forum. The author of this post is the same developer who wrote the FailPics! web application. You can safely assume that the database and user settings displayed here are not the same as those used on the target application.

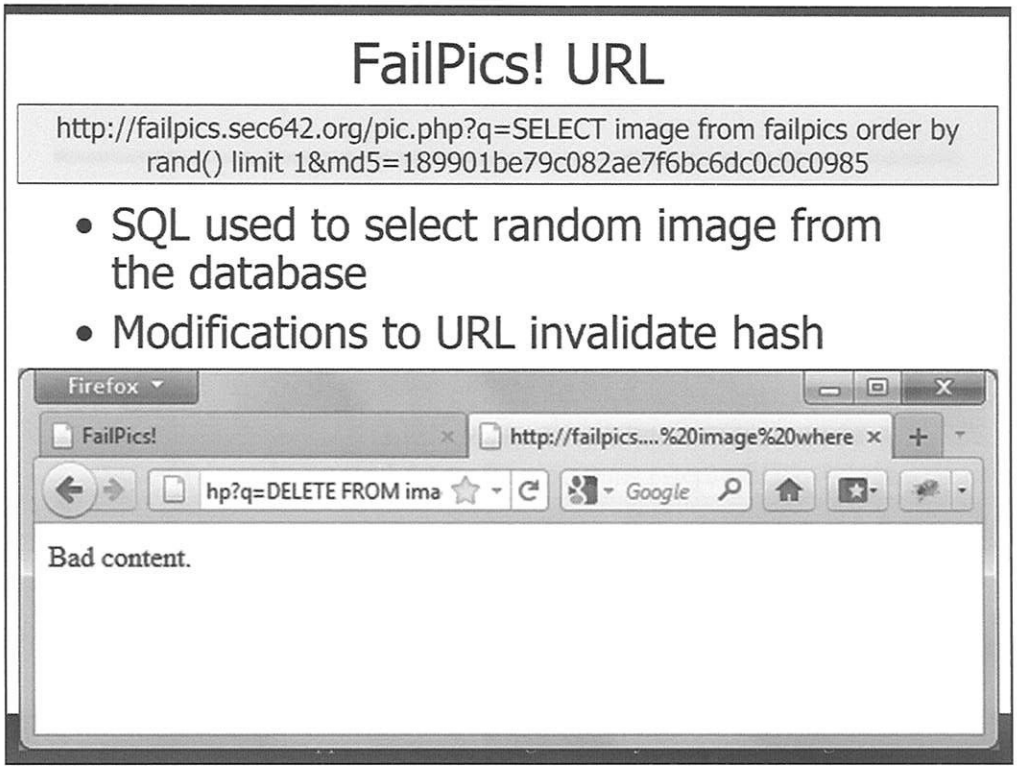


Weak Key Attack – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.

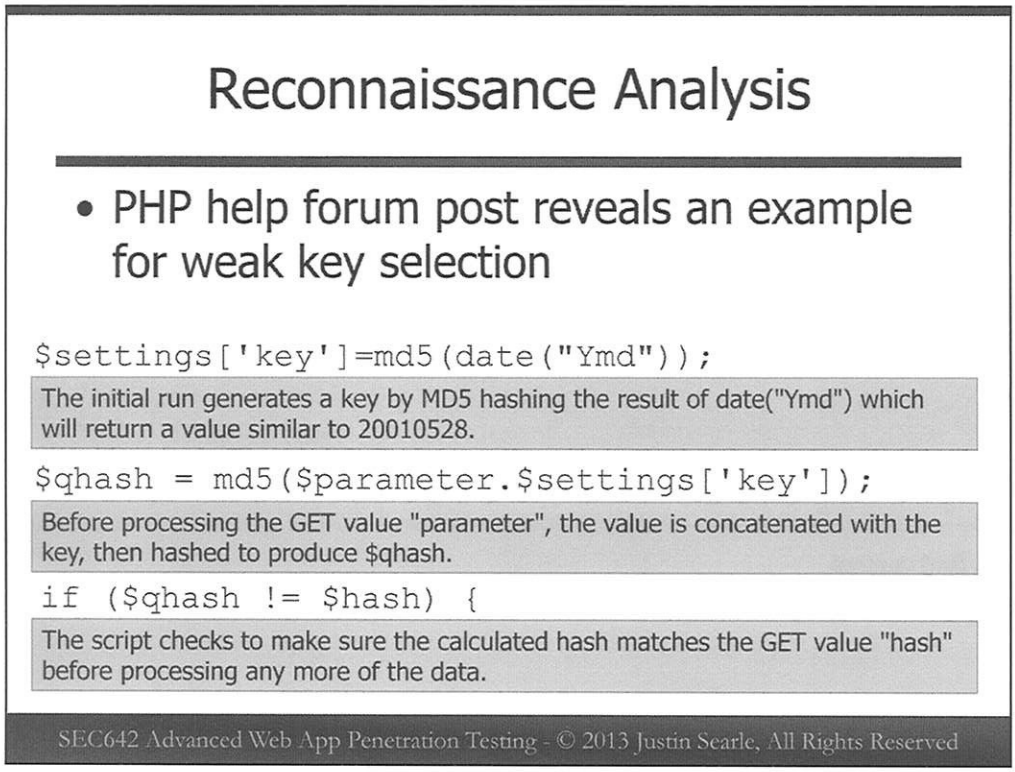


In this lab exercise you'll target the weak key selection for the FailPics! site at http://failpics.sec642.org. At the main page, click the "Click for Fail!" button to open a new window that includes a randomly selected image.



FailPics! URL

Examining the FailPics! URL we see that it is using a SQL statement to randomly select an image for display from the database. If we attempt to change the SQL however, we get a "Bad content." error message. Note that one of the GET parameters is an MD5 hash.



Reconnaissance Analysis

The PHP help forum post reveals interesting techniques used by the developer of the FailPics! application for key selection and hash computation.

```
$settings['key']=md5(date("Ymd"));
```

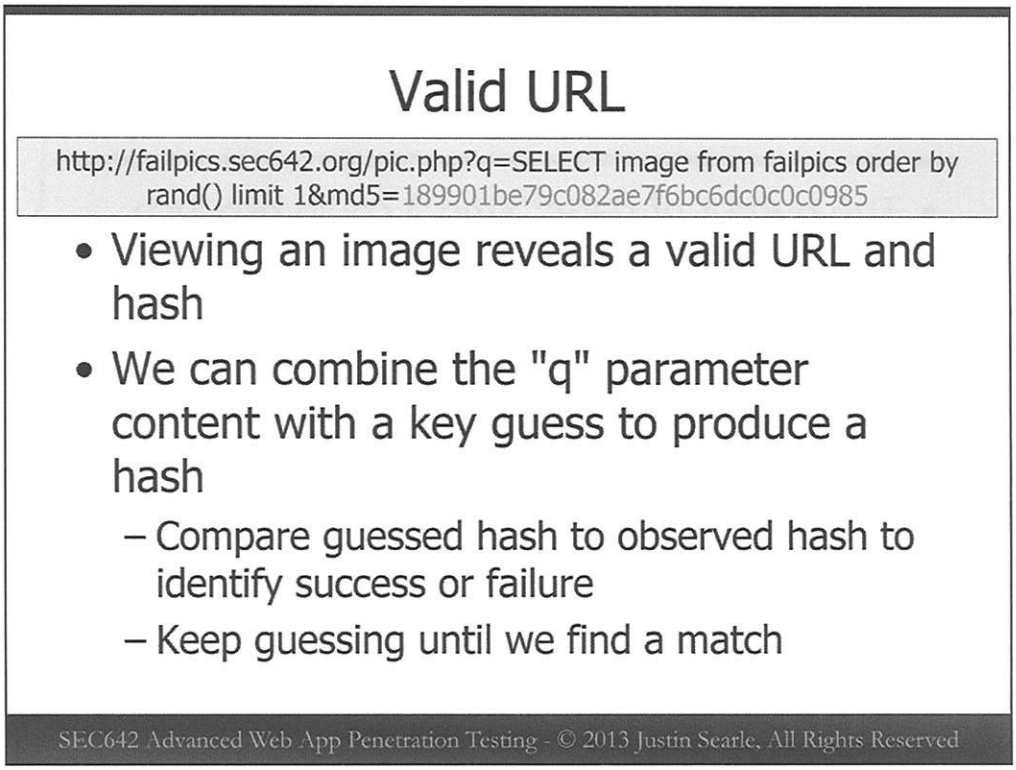
This code sets the key member of the $settings available to the MD5 hash of the date() output. Invoked with the settings "Ymd", the date() function will return a value similar to "20010528" where the year is 2001, the month is May and the day is the 28th. Note that this code is only executed if the "$NEWINSTALL" flag is set, indicating that it is only set at installation or first-run. More information on the PHP date function can be found at http://php.net/manual/en/function.date.php.

```
$qhash = md5($parameter.$settings['key']);
```

This code sets the variable $qhash (possibly, "query hash"?) to the MD5 output of the value $parameter concatenated with the key value generated in the previous step. Note that the $parameter value was taken from the $_GET parameters earlier in the code.

```
if ($qhash != $hash) {
```

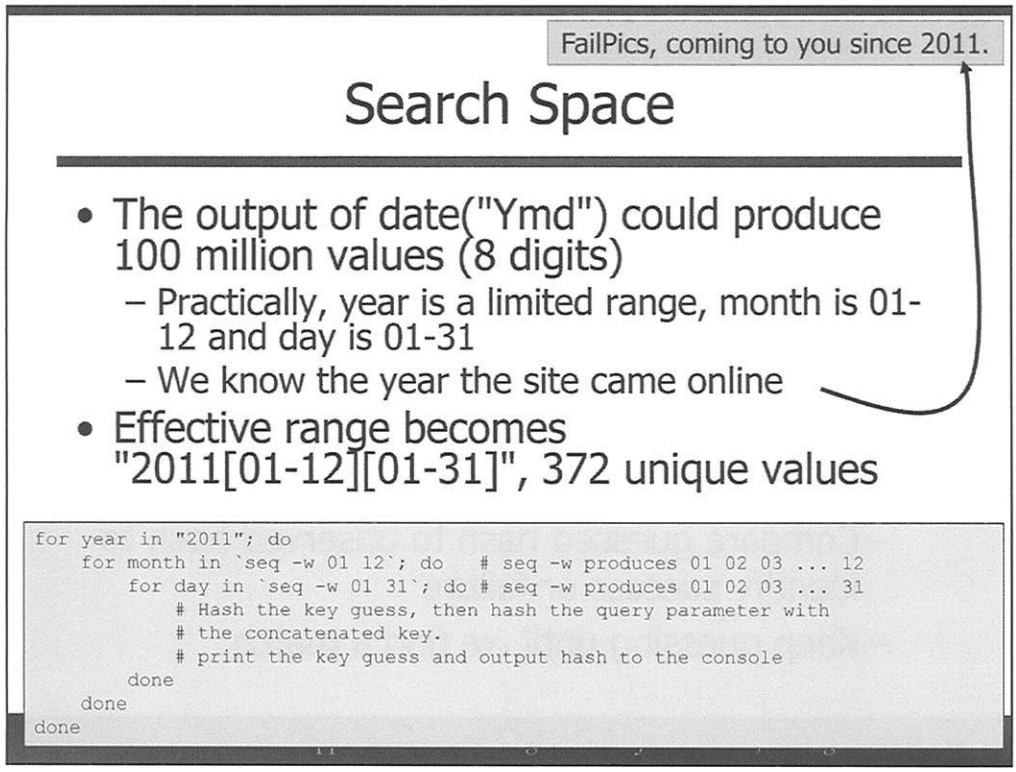
This code compares the hash of the URL parameter, including the key, to the query hash included in the URL to ensure they are the same before processing the data.



Valid URL

This slide shows a valid URL from the FailPics! application. The "q" parameter is a SQL SELECT statement, using the rand() function and limit 1 constraint to select a random image from the failpics table. Leveraging what we learned from the PHP forum post, we know that the MD5 hash in the URL is the result of MD5 hashing the concatenation of the SQL query and the key.

If we guess the key repeatedly, hashing the valid SQL statement with our key guess, we can compare our guess hash to the observed hash. If they match, we know we have guessed the correct key. If they do not match, we keep guessing. Fortunately for us, the search space for the key is limited.



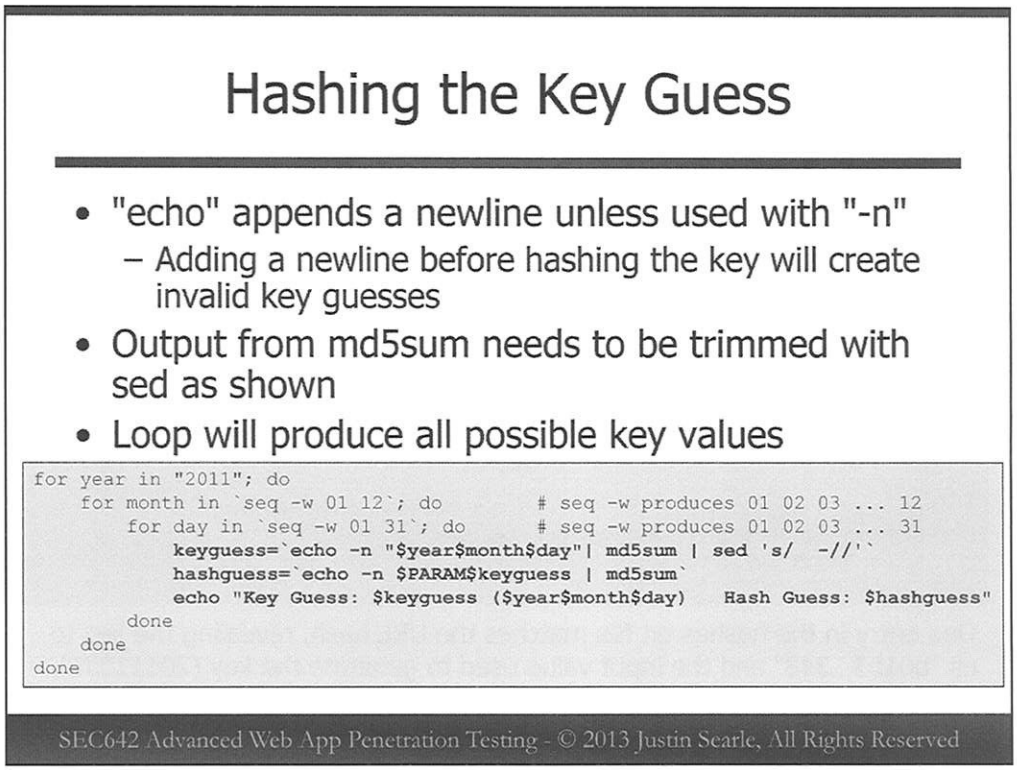
Search Space

If we assume that the key was generated with an MD5 hash of the output of the PHP date("Ymd") function, we might think the search space is 100 million values from 00000000 to 99999999. However, we know that the entropy is more limited, since only valid months (01-12) and days (01-31) are used. Further, the year value can also be considered to be of limited entropy for our calculation.

Visiting the FailPics! site, we see a tagline on the bottom, "FailPics, coming to you since 2011". Leveraging 2011 as our base year, we reduce our search space to 372 unique values (without taking into consideration months that have fewer than 31 days).

The Bash script excerpt in this slide uses three for loops to iterate through all the possible values. The initial for loop is unnecessary as it iterates through a single value "2011". The second for loop iterates through the month values 01 through 12 using the seq command (seq with the -w argument preserves the leading 0 for months before October, needed for our attack), and the third loop iterates through the days of the month 01 through 31.

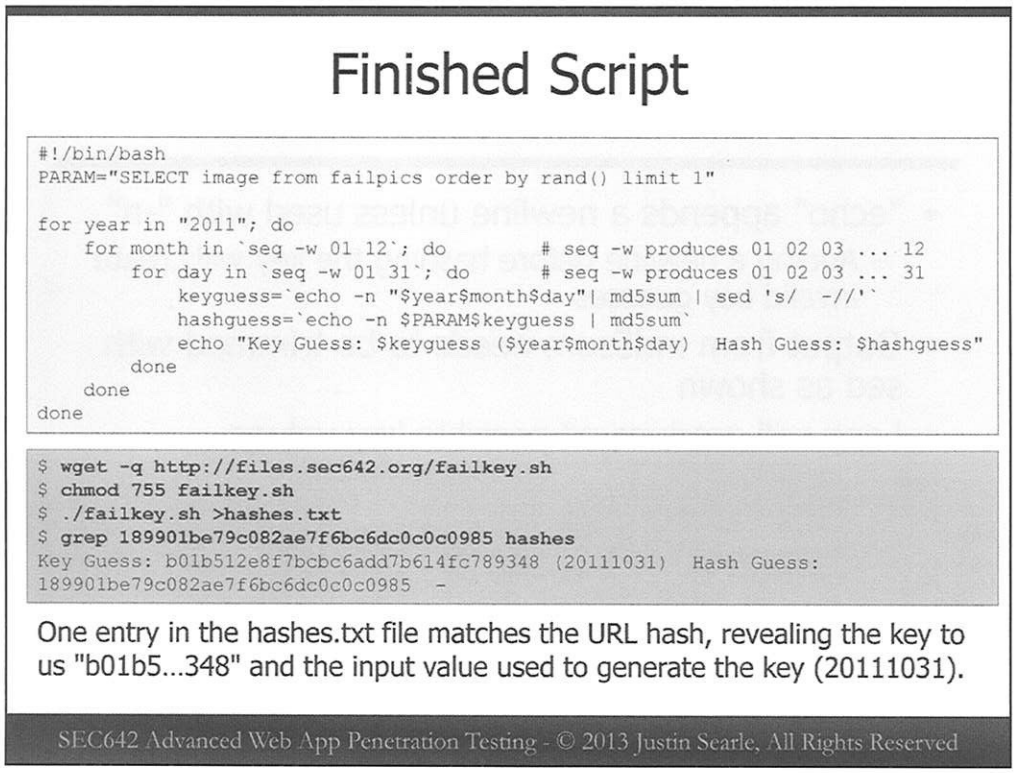
With the three for loops in place, we need to add the code to compute the hash using the key guess and print the key guess and output hash to the screen.



Hashing the Key Guess

The three added lines in bold above are used to round out the script. Here, the concatenated year, month and day values are generated and sent to the md5sum command-line utility to generate the MD5 hash (we use "echo -n" to suppress the automatic newline the echo utility adds, an important step since adding a newline would invalidate our MD5 hash for key guessing). The sed command strips off the " -" added to the end of the md5sum output.

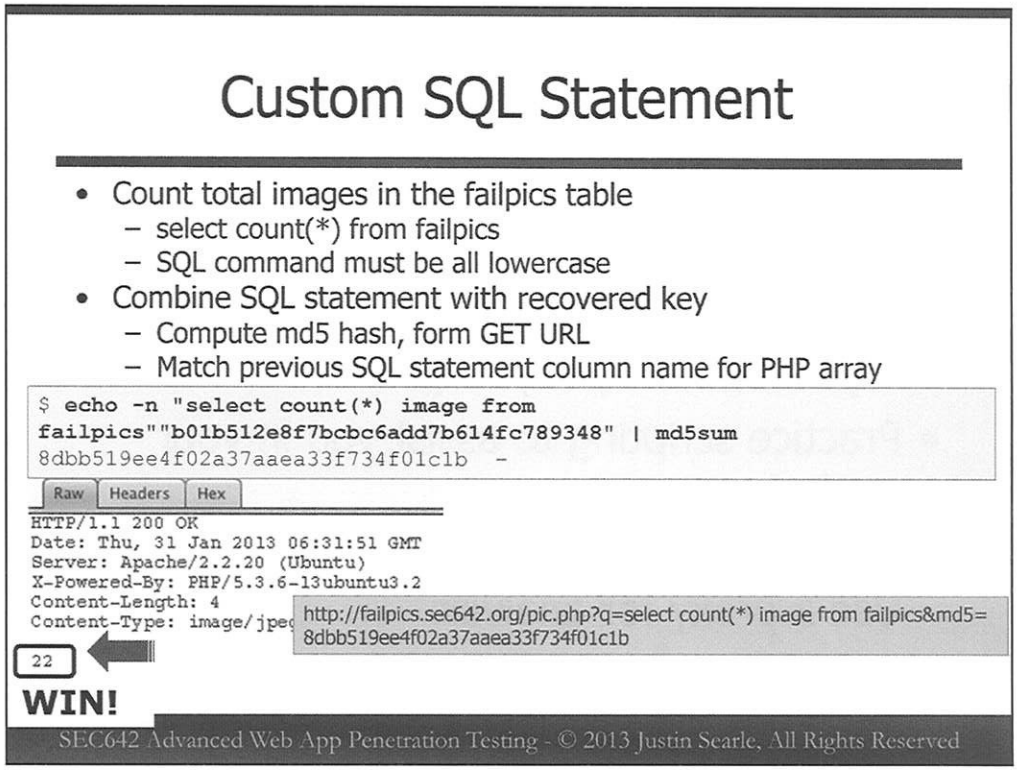
Next, our key guess is used to compute a hash guess, using the SQL parameter set in the $PARAM variable (we'll set that variable in the next slide). Finally, we echo the output of the key guess (including the year+month+day input) and the hash guess to the screen.



Finished Script

The finished script is shown on this slide, taking the SQL statement from the FailPics! site to populate the PARAM variable. You can download this completed script from http://files.sec642.org/failkey.sh.

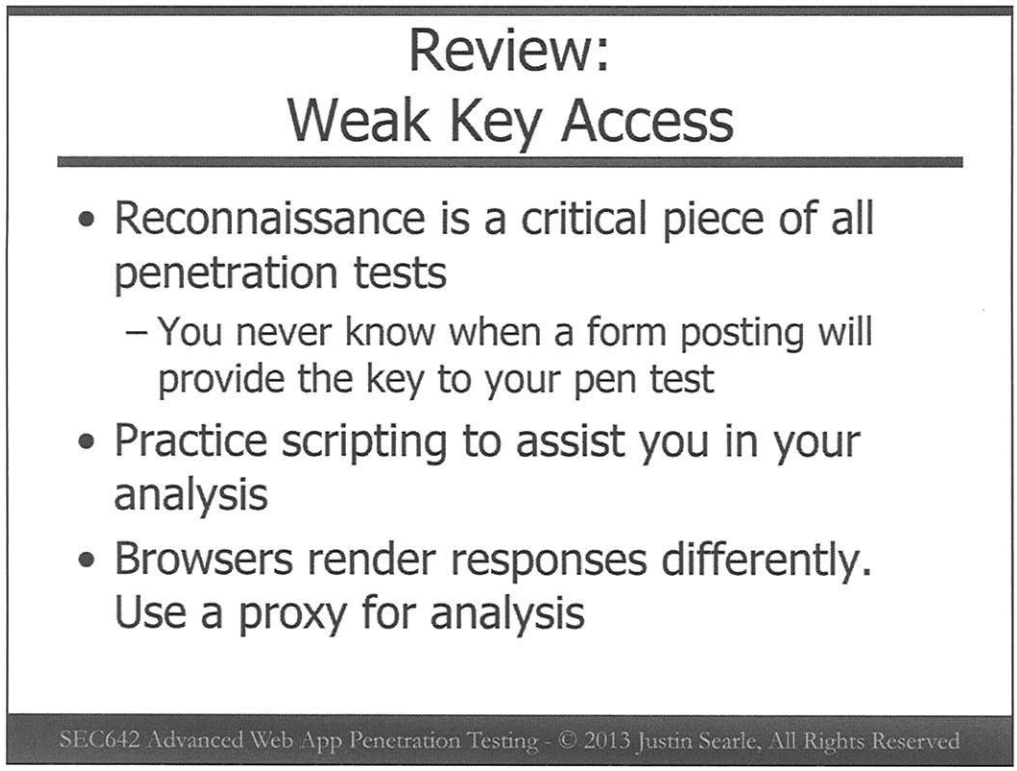
Running the script and redirecting the output to a file "hashes.txt", we can search for the key guess that matches the valid HASH from the FailPics! site. This reveals one matching hash, disclosing the correct key guess and hash.



Custom SQL Statement

With the recovered key, we can produce our own SQL statements by creating valid hashes. Our SQL statement to count the number of entries in the table is shown in this slide. We use "echo -n" again here to suppress a newline character, concatenating the new SQL statement with the valid key and hashing the results. The output hash "8dbb51…" is then used to create a valid FailPics GET query, revealing the output of the SQL statement as shown in Burp's Proxy history.

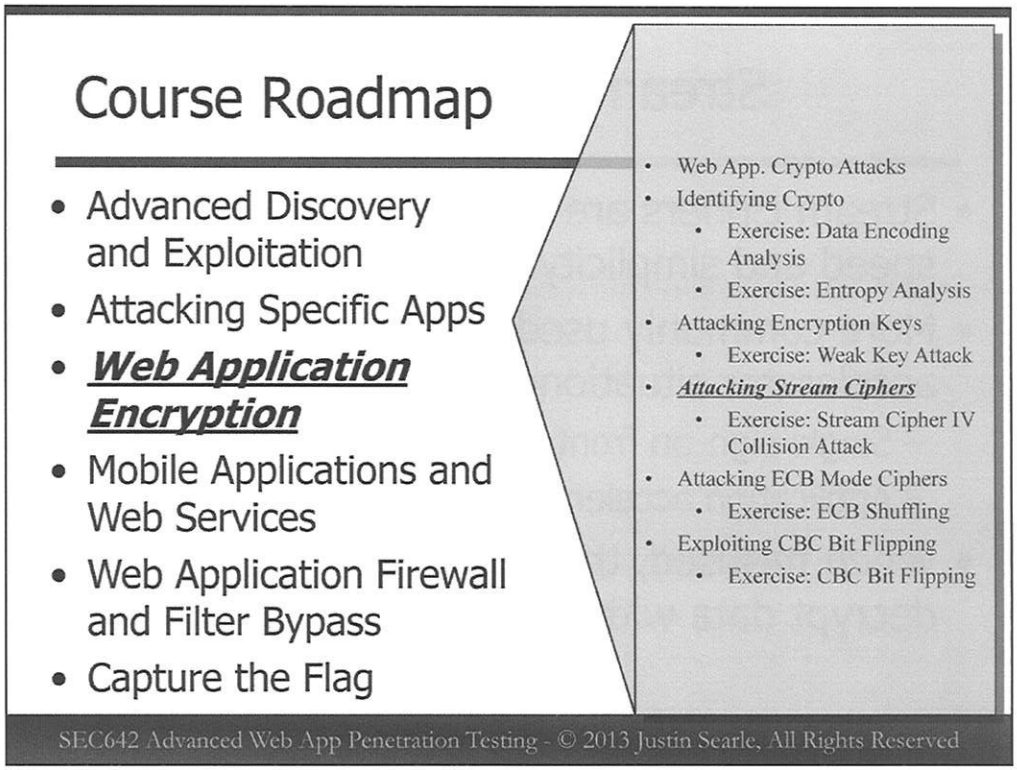
If you are attempting to use Burp Repeater in this exercise, make sure that all spaces are replaced with their URL encoded version `%20`.



In this exercise we found an obviously bad piece of coding which put the database query in the web pages URL. However, the MD5 hash in the second parameter could have derailed us. It did not though, due to two key mistakes by the developer. First is that the method of creating the key was extremely weak. With only 365 days in a year, the application is depending on a fairly small and predictable range of numbers as the seed of the key. The second was posting this method to a forum posting to ask for help. Our reconnaissance told us exactly what we needed to know to break the method create the MD5 parameter value so that we could then run arbitrary SQL on the database. One of the key lessons here is never to short change your reconnaissance. It frequently makes the difference between a successful penetration test and one that leaves us stuck.

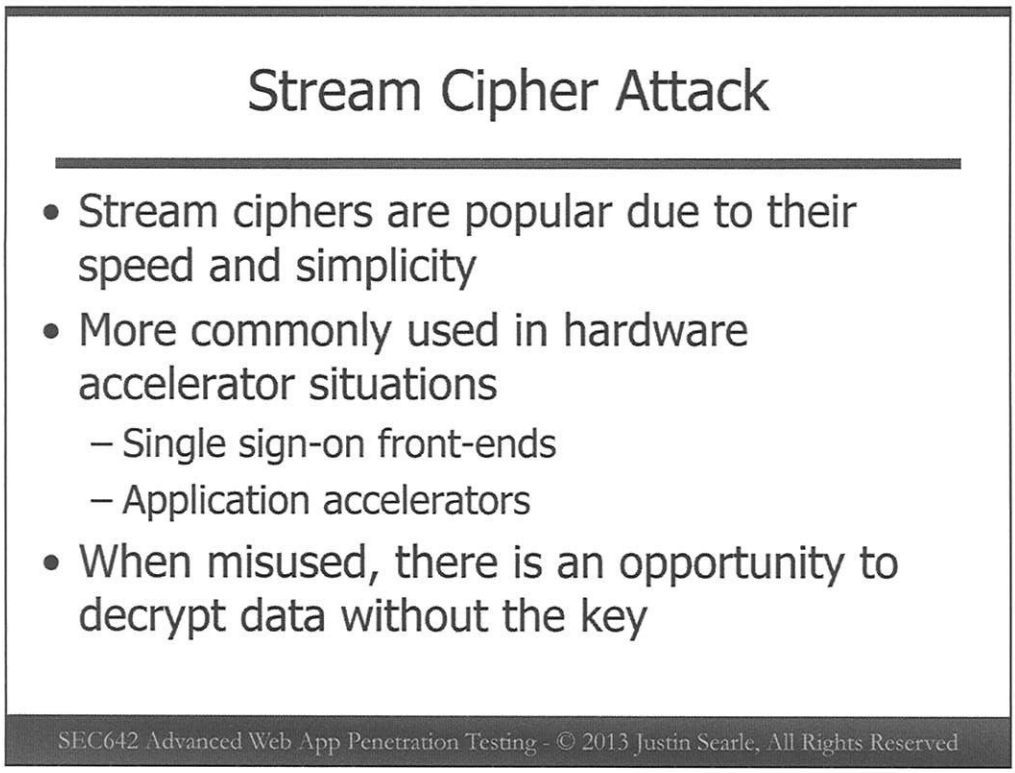
The next lesson is that we need to practice scripting as often as we can. This exercise required a script to break the key to this web page. Our time during penetration tests is limited and it is critical to be able to put together scripts in as short a time as possible. So practice scripting in your language of choice and be prepared to use it during your engagements.

Last, when we create our own database query and MD5 hash and send it to the web server, the browser may or may not render the response in something visible to us. The query we created pulled the number of images stored in the database and sent that back to the browser. However, only Internet Explorer will actually show the resulting count. We needed to review the response in Burp to see that there were 22 image entries in the database. Do not get too dependent on your browser as your main analysis tool, but instead focus on your favorite interception proxy. This removes any issues with how a browser decides to render the response to our attacks.



Course Roadmap

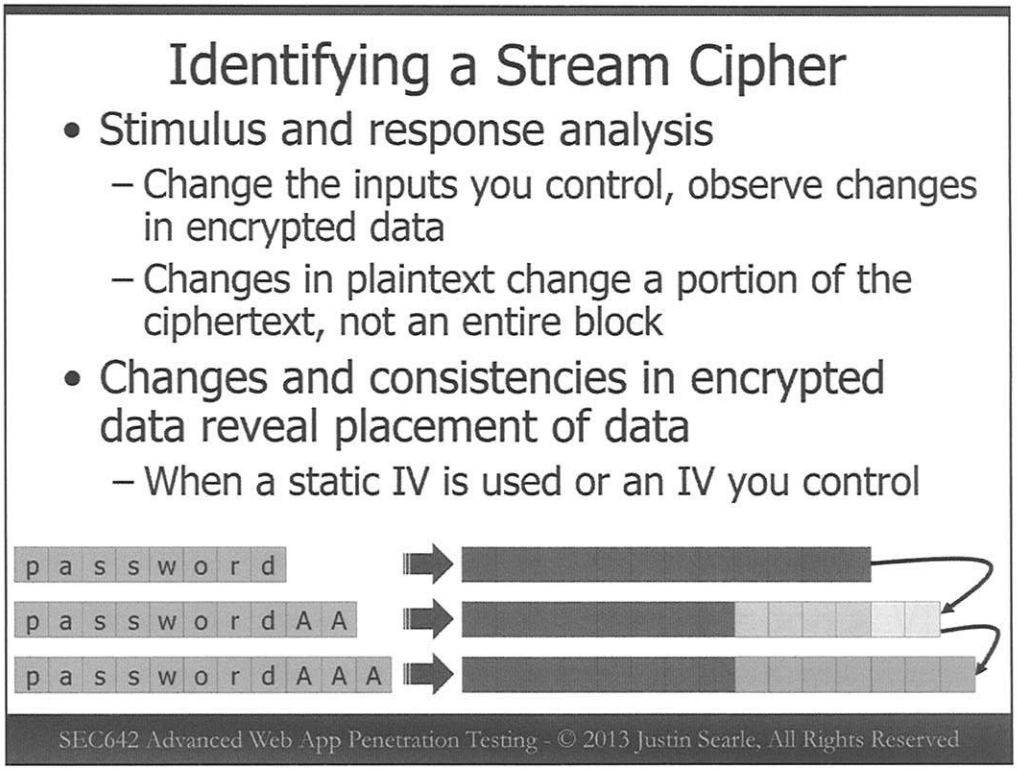
Next we'll look at exploiting web application systems using stream ciphers.



Stream Cipher Attack

Stream ciphers have historically been favored for their speed and simplicity in deployments, particularly in their adaptation to hardware implementations. For this reason, stream ciphers are often found in hardware-based cryptographic accelerator devices which may provide a level of security beyond that of a single web application. Commonly, stream ciphers will be used in single sign-on front-ends for web applications, as well as application accelerator platforms.

A stream cipher can be a secure encryption method, but the simplicity of a stream cipher can be misleading, leading to implementation weaknesses that can allow an attacker to decrypt data without knowledge of the encryption key.



Identifying a Stream Cipher

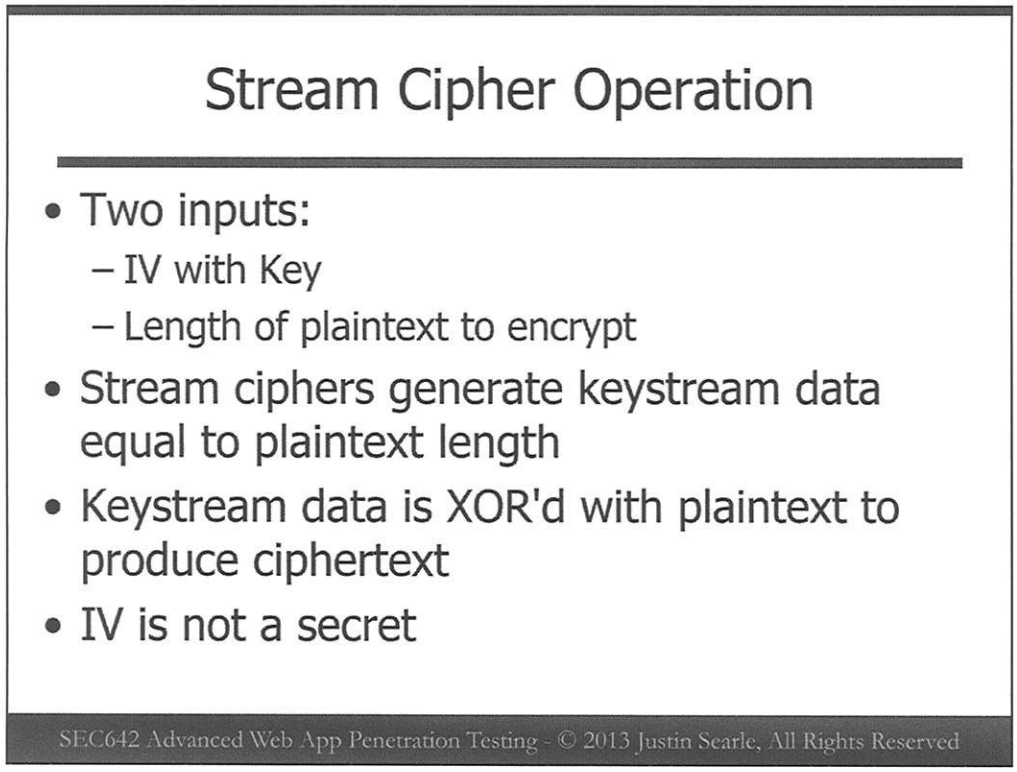
Remember that a stream cipher encrypts one bit of data at a time, so the length of the ciphertext is equal to the length of the plaintext (plus the possibility of header information or checksum data, which would typically be fixed-length). In order to evaluate and identify a stream cipher, we can use stimulus and response analysis.

For some controlled plaintext input into the encryption algorithm, create and record the ciphertext for corresponding plaintext values. In the example on this slide, we initially encrypt the string "password", which produces an output value of 12 bytes (our 8 byte string and 4 bytes of unknown content). Next, we change the string "password" to "passwordAA". The resulting encrypted value is 2 bytes longer, corresponding to our additional two "AA" bytes. This confirms that we are looking at a stream cipher.

Further, our ciphertext remained consistent for the first 8 bytes of the encrypted value. This behavior reveals two facts about the cipher implementation:

1. The string "password" is at the beginning of the encrypted content, with unknown ciphertext at the end.
2. A static IV is used for the encryption of both values, since the output ciphertext is consistent for similar plaintext across multiple encryption operations.

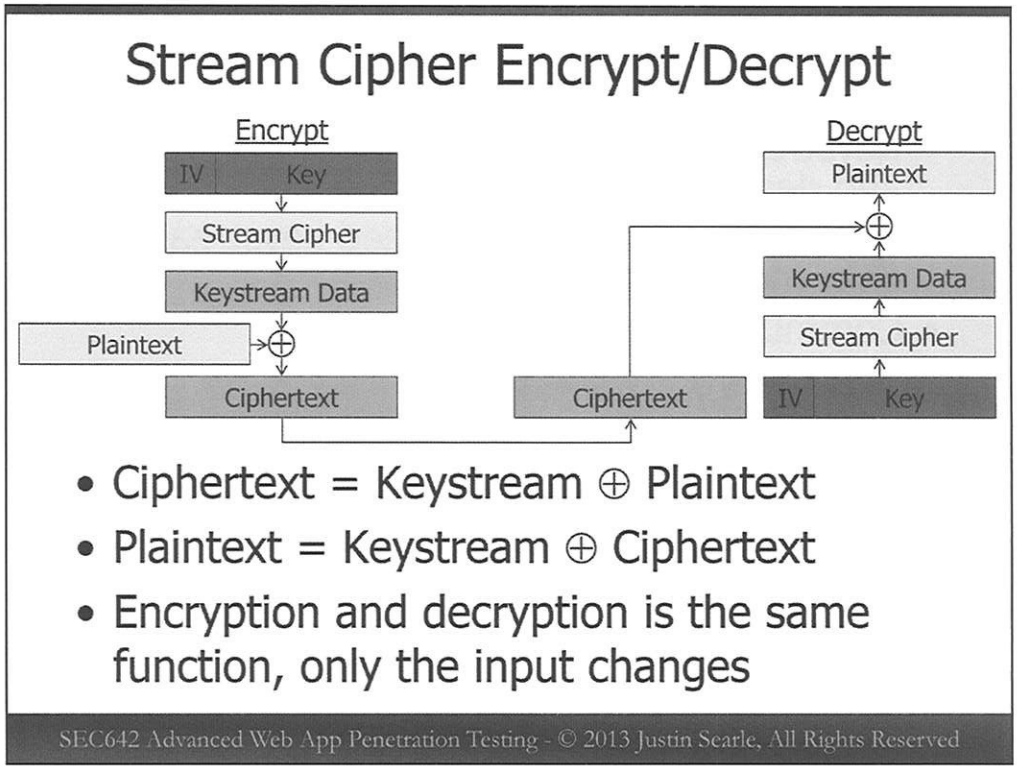
If we can consistently repeat this process (as in the third plaintext string of "passwordAAA"), then we have discovered an exploitable vulnerability in the operation of the cipher.



Stream Cipher Operation

As a short refresher, recall that a stream cipher accepts two inputs: the IV concatenated with the encryption key and the length of plaintext to encrypt. The stream cipher generates keystream data as the output of the algorithm, equal to the length of the plaintext to be encrypted. The keystream data is XOR'd with the plaintext to produce ciphertext.

The IV itself is not a secret, but must not be repeated with the same key to preserve the security of the cipher.



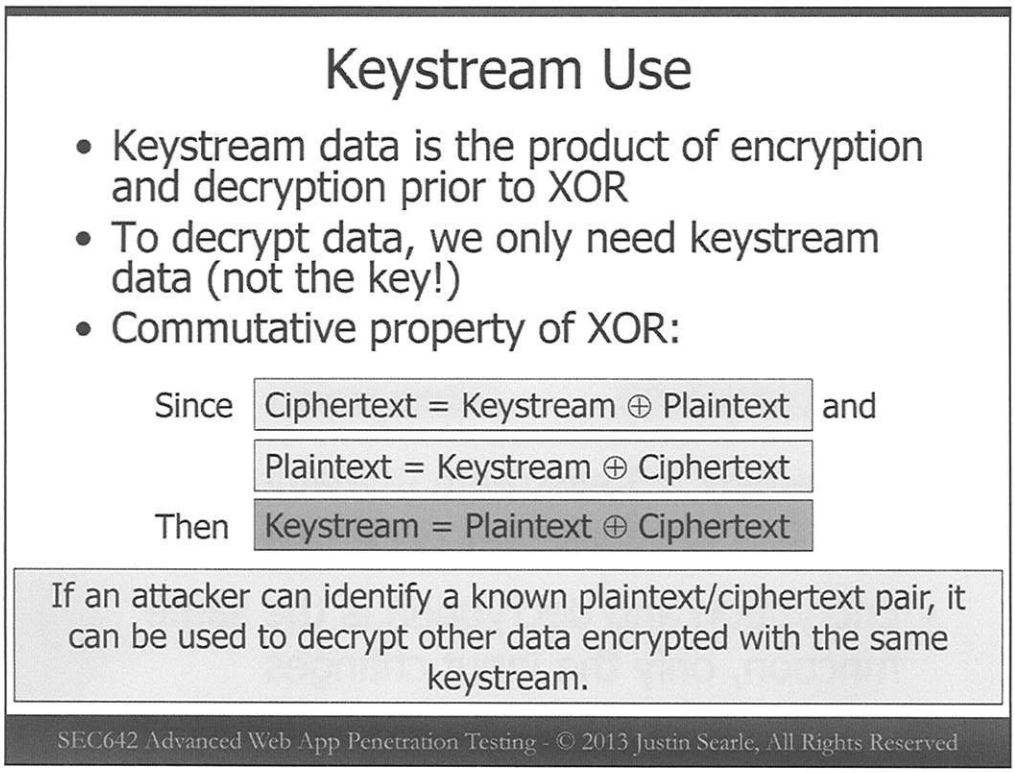
Stream Cipher Encrypt/Decrypt

This slide illustrates the operation of encrypting (on the left) and decrypting (on the right) ~~a~~ with a stream cipher. For encryption, the IV and the key are concatenated and sent to the stream cipher to produce keystream data. The keystream data is XOR'd with the plaintext to produce ciphertext.

The ciphertext is then transmitted over the network. The IV may be transmitted with the ciphertext (and commonly is, if the IV is not a static value).

Upon receiving the ciphertext packet, the IV and key are concatenated and then passed to the cipher to create keystream data. The keystream data is then XOR'd with the ciphertext to produce plaintext.

Looking at this procedure, we see that the ciphertext is produced by XOR'ing the keystream data with the plaintext. Conversely, the plaintext is produced by XOR'ing the keystream with the ciphertext. The encryption and decryption processes are identical, the only difference is the input value as ciphertext or plaintext to produce plaintext or ciphertext.

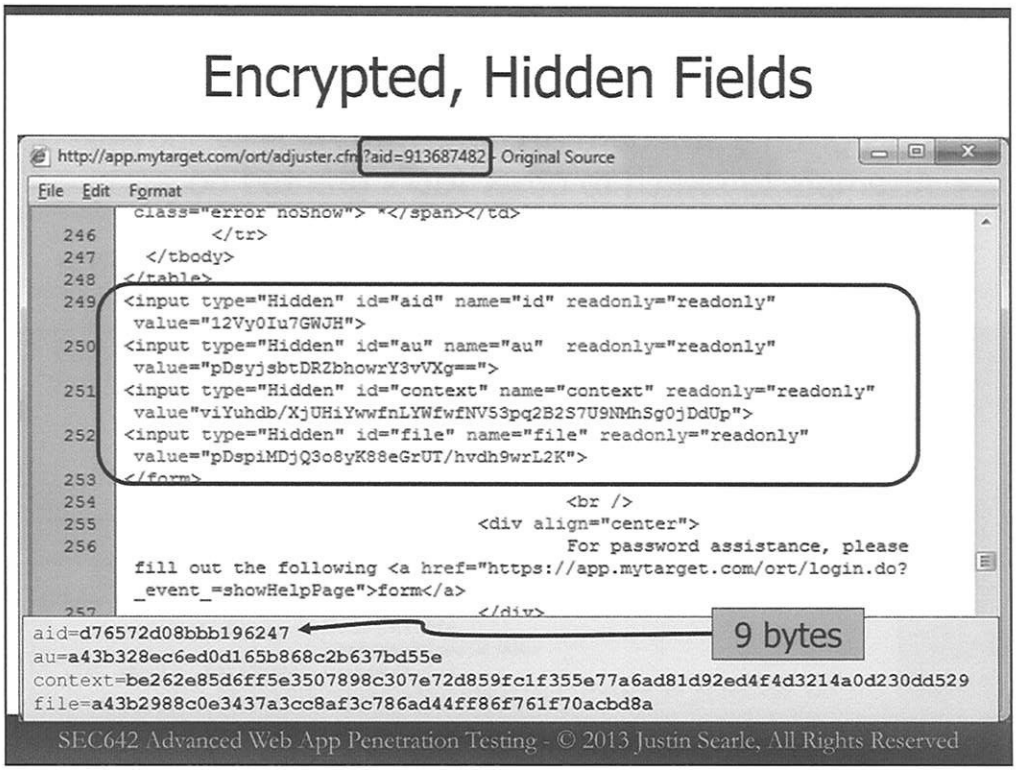


Keystream Use

We examined the result of XOR'ing plaintext with keystream, and ciphertext with keystream. When we XOR plaintext and ciphertext however, we reveal the original keystream that was used to encrypt the data. This is possible through the commutative property of XOR.

If we want to decrypt data in an attack, it is not always necessary to know the encryption key itself. Since the key only generates keystream data, if we know the keystream data then we can simply XOR keystream with ciphertext to produce plaintext. In order to obtain the keystream data however, we need to both observe an encrypted value, and to know the plaintext that was used to produce the ciphertext. This assumes some level of control over the encrypted content, or other opportunities for an attacker to identify encrypted values when the input plaintext is already known.

In order to help illustrate this attack, let's look at a sample case the instructor examined during a penetration test. Note that this example has been obfuscated to protect the identity of the customer with the vulnerable application.

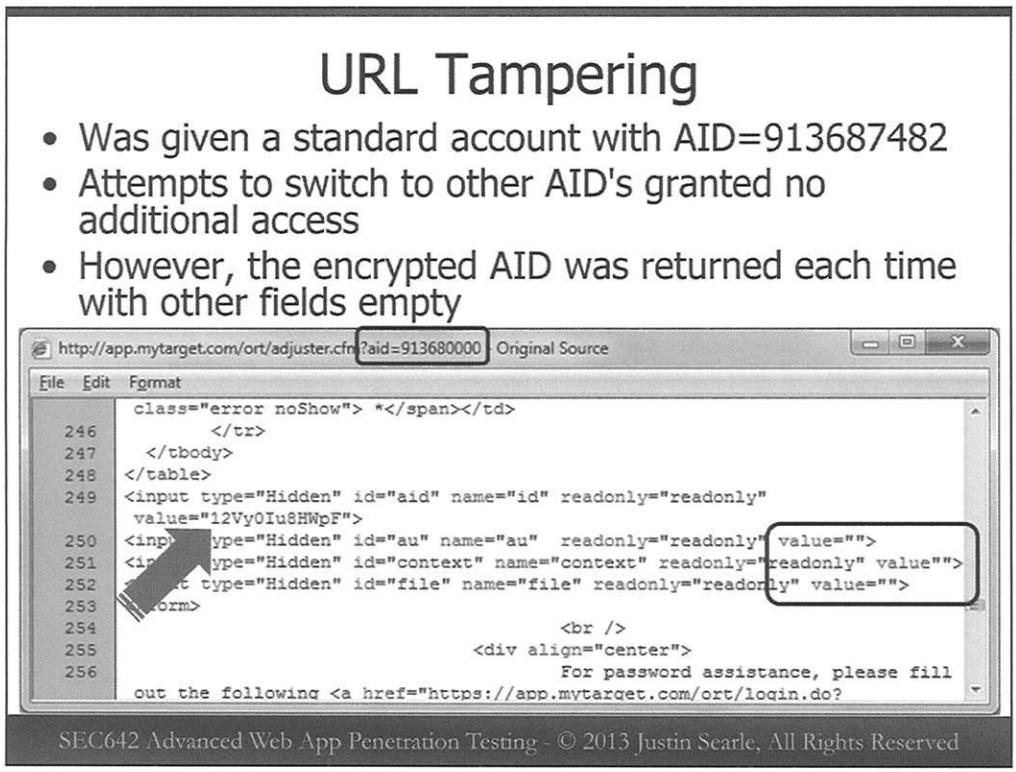


Encrypted, Hidden Fields

In the analysis of encrypted data in a ColdFusion application, several hidden fields were identified as part of a form page. This portion of the application was only accessible after logging-in with credentials supplied by the customer.

The hidden fields shown in this slide "aid", "au", "context" and "file". Looking at the length of the encrypted fields we commonly see odd-length values, indicative of a stream cipher.

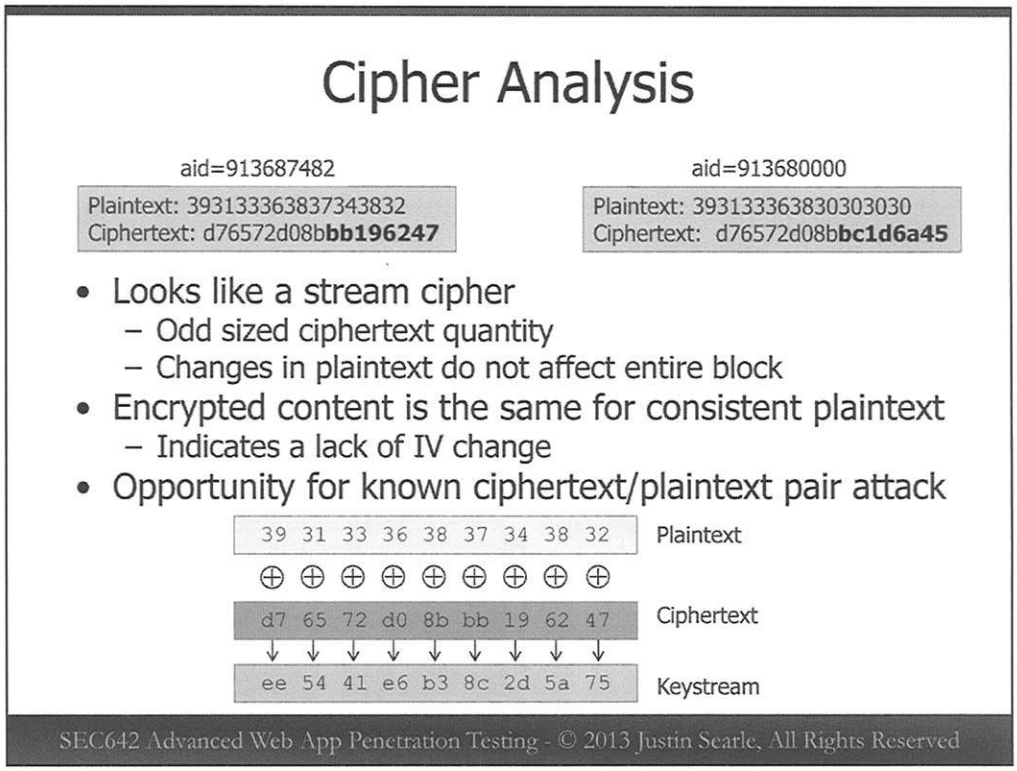
Note that, in the URL bar, a GET parameter field "aid" is present. The numerical value 913687482 is the numeric identifier assigned to the test account supplied to us during the penetration test.



URL Tampering

In an earlier test we tampered with the URL bar and attempted to access the resources of another user without having the alternate user's login credentials. Upon modifying the URL however, we were no longer granted access to the site. Our hidden form fields changed, however, making the values of "au", "context" and "file" empty. Note that the encrypted value of the "aid" field is still present.

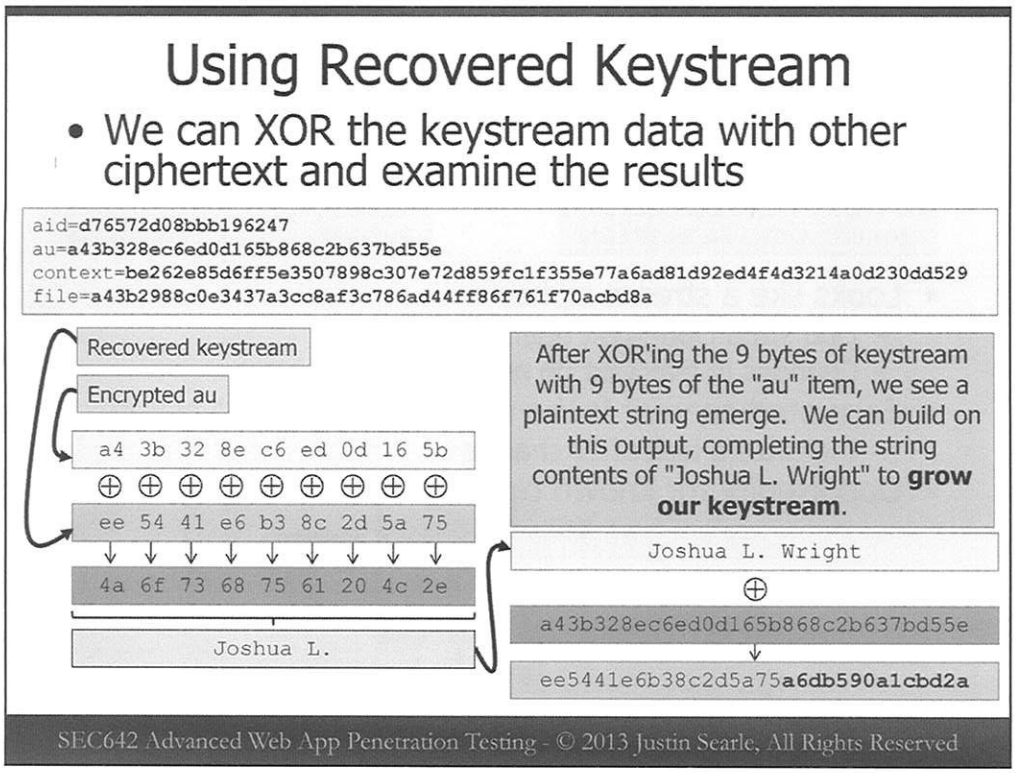
Our best guess at the time of the assessment was that the "aid" in the URL corresponds to the encrypted "aid" in the form body and is used for some sort of authenticated session validation.



Cipher Analysis

Examining the encrypted and plaintext of the "aid" element discloses interesting properties about the encryption used for the application. First, note that the ciphertext values corresponding to the initial and second "aid" values are very similar; the ciphertext values are identical up to the 6th byte, corresponding to the first changed byte in the plaintext. This was sufficient for us to confirm that the encrypted "aid" value in the form represented the plaintext "aid" element in the URL. This encrypted value behavior, and the encrypted "aid" length of 9 bytes also confirmed for us that a stream cipher was used. Finally, the similarity in encrypted content between the two encrypted "aid" values showed us that a static IV was used (if a non-static IV was used, the ciphertext for the two encrypted values would be completely different, even if the plaintext values were similar).

With these requirements met we have an opportunity for a known ciphertext/plaintext pair attack. By XOR'ing the 9-byte plaintext "aid" with the encrypted version, we can produce keystream data, as shown on this slide.



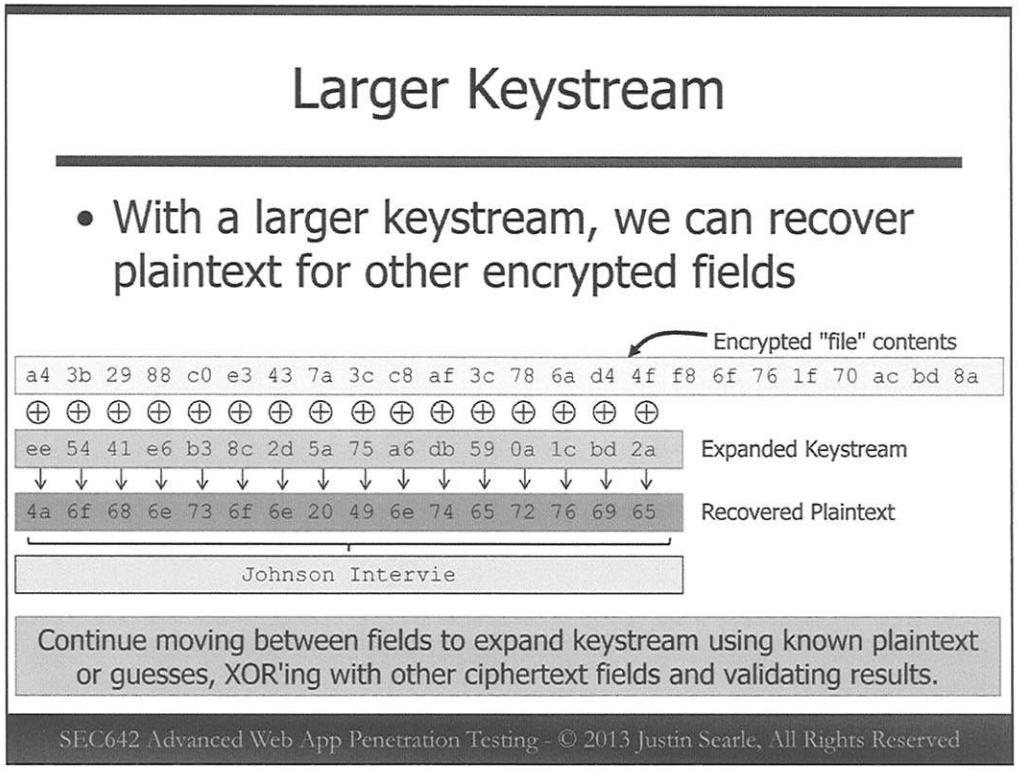
Using Recovered Keystream

With 9 bytes of recovered keystream data we can attempt to decrypt the other form elements we saw initially such as "au", "context" and "file," including up to the first 9 bytes of each. Unfortunately, we are limited in our ability to decrypt longer values since our known plaintext/ciphertext pair was only 9 bytes in length.

On the left of the slide we take the encrypted "au" value from the form and XOR it with the 9 bytes of recovered keystream data. The output value is an ASCII string, consisting of "Joshua L.".

While our keystream length is limited, we can attempt to extend it through the recovery of other plaintext. The "au" field revealed the plaintext value "Joshua L." which was the logged-in user's identity. Since we know the last name is Wright, we can deduce the entire 16-byte value in plaintext is "Joshua L. Wright". We can then use the encrypted form of "au" and fully-recovered plaintext string to produce more keystream data, shown on the right of the slide.

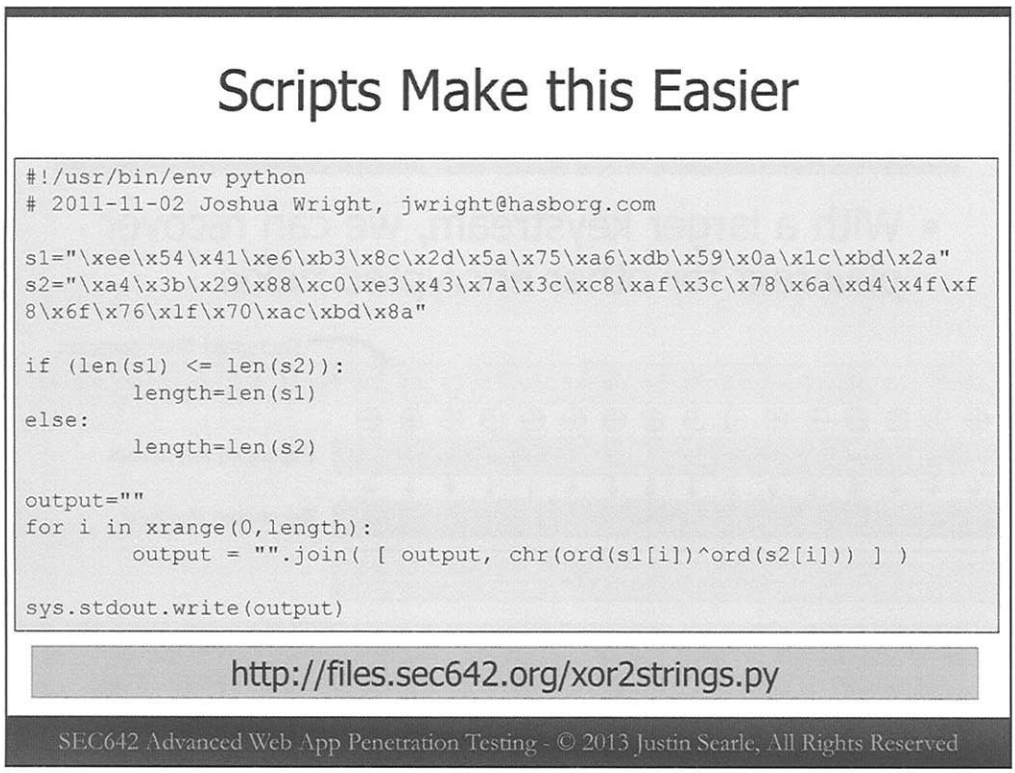
We started out with 9 bytes of keystream, but upon decrypting other fields we were able to deduce additional plaintext for known ciphertext, allowing us to grow our keystream value. With 16 bytes of keystream, we can attempt to recover the plaintext for the longer "file" or "context" fields next.



Larger Keystream

Using the larger keystream data recovered from the expanded "au" encrypted content, we can move on to decrypting the "file" element. In this case, we can recover the first 16 bytes of the 24 byte "file" element, revealing the string "Johnson Intervie". We can recover an additional byte of keystream here by adding the trailing "w" to "Intervie", though the remainder of the plaintext remains a mystery. Next we can apply the same technique to the "context" field, hopefully recovering sufficient keystream to decrypt the remainder of the "file" content.

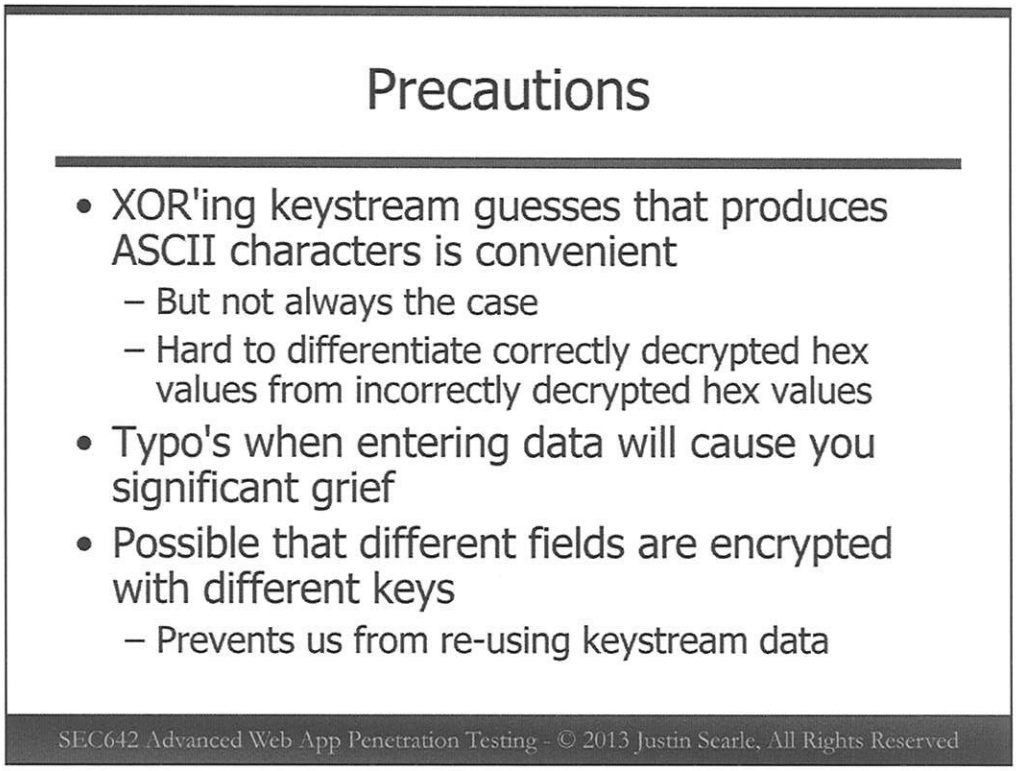
In this test we were able to keep growing the keystream to sufficiently decrypt any encrypted values up to 128 bytes in length. With a keystream value of this size, most fields were easily decrypted, giving us an opportunity to exploit a local file inclusion (LFI) flaw on the system.



Scripts Make this Easier

The simple Python script in this slide takes any two strings in hex-encoded notation and XOR's them together up to the length of the shorter of the two strings, displaying an output value on the screen. Using this script we can recover keystream by setting the two strings be plaintext and ciphertext, or decrypt a value by setting the two strings to keystream and ciphertext.

This script is available on the SEC642 file download server at the URL shown on this slide.



Precautions

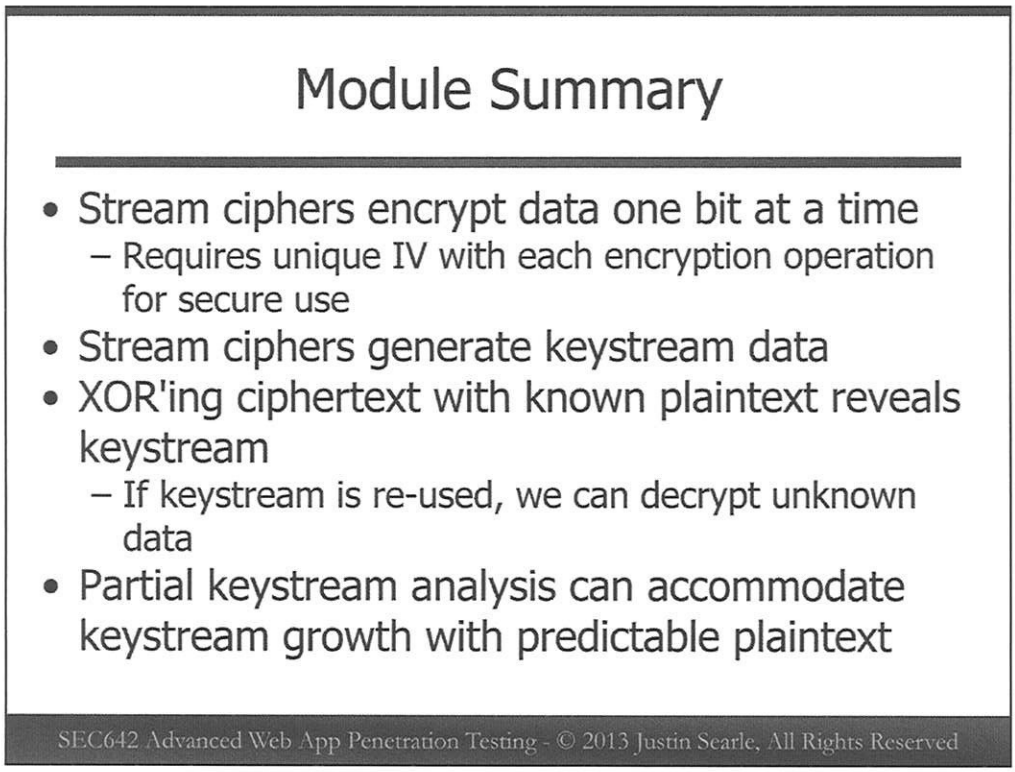
In the example we examined as part of this content, it was easy to identify when the XOR product of keystream and ciphertext produced a valid value, since the output values were all plaintext strings. This is not always the case, however, and it can be difficult to know if the decrypted value is valid. Look for common binary data encoding mechanisms, such as a length field followed by a corresponding number of bytes of content before a second length field is identified, or common binary data values such as the NULL byte (0x00).

Also be cautious about entering the strings to XOR; a typo or an omission in one of the strings will invalidate your results.

Remember that multiple conditions must be present to successfully exploit a stream cipher in this fashion:

- Static IV or common IV collisions with a small IV space;
- Known ciphertext and plaintext pairs;
- Values are all encrypted with the same encryption key.

If these conditions are not met, it will not be possible to leverage keystream data to decrypt other elements.

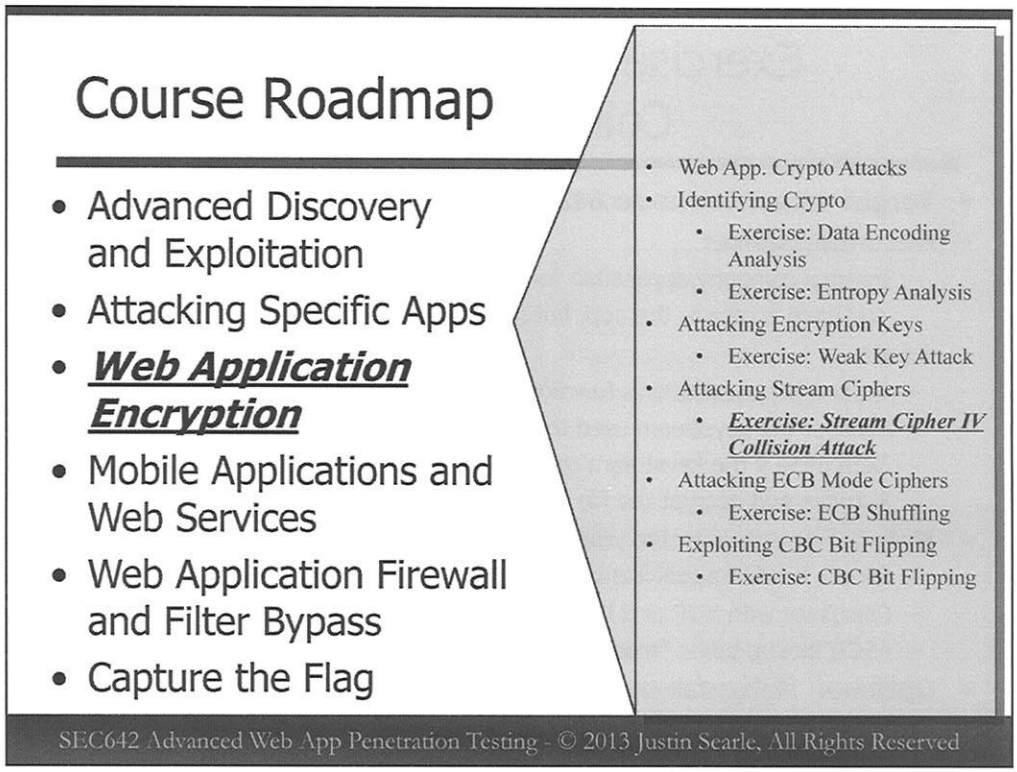


Module Summary

In this module we examined the common property of stream ciphers, encrypting one bit of data at a time. In order to use a stream cipher securely, a unique IV is needed each time data is encrypted. However, this practice is commonly omitted in application implementations as it requires an added level of complexity for the transmission of IVs and rotation of encryption keys when the IV space is exhausted.

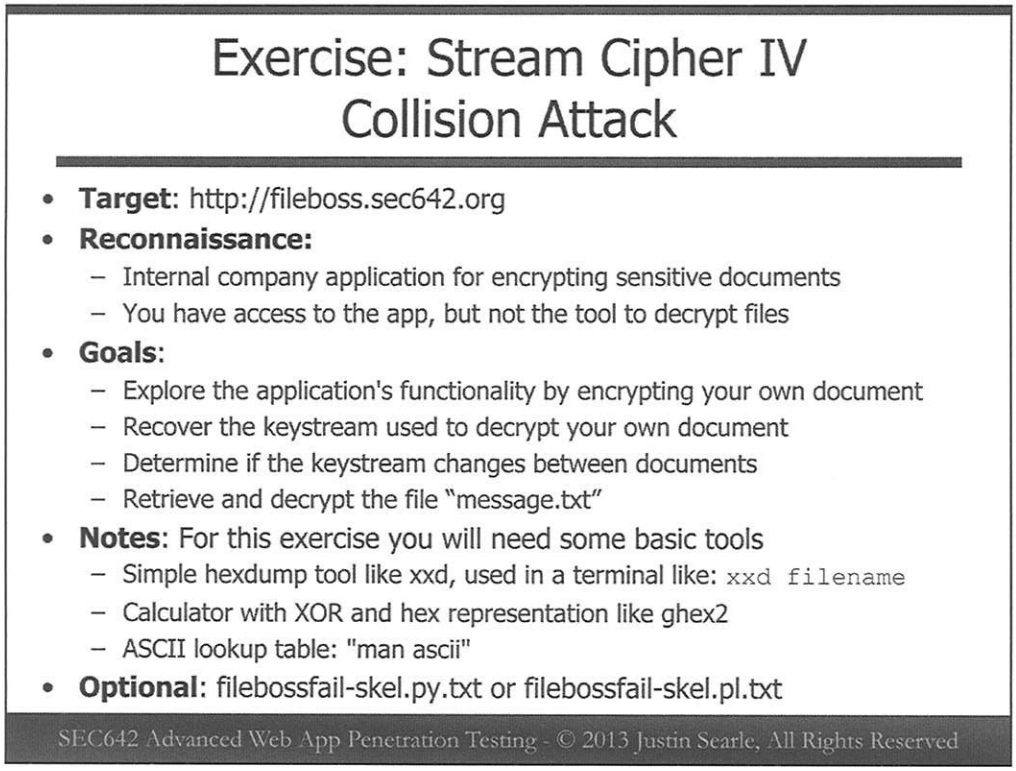
Stream ciphers generate keystream data, using the keystream data to encrypt or decrypt data by XOR'ing it with the plaintext or ciphertext value. When we XOR the ciphertext with known plaintext however, we recover the keystream data. If the keystream data is re-used (e.g. a static IV) then we can also use it to decrypt other unknown ciphertext values.

When recovering keystream data from known plaintext and observed ciphertext, we may not have sufficient keystream to decrypt all the data. In some cases we can increase the keystream by decrypting values and extrapolating the remaining unknown plaintext bytes. With a longer plaintext value we can grow the keystream to decrypt other encrypted elements.



Course Roadmap

Now we will do an exercise.



Exercise: Stream Cipher IV Collision Attack

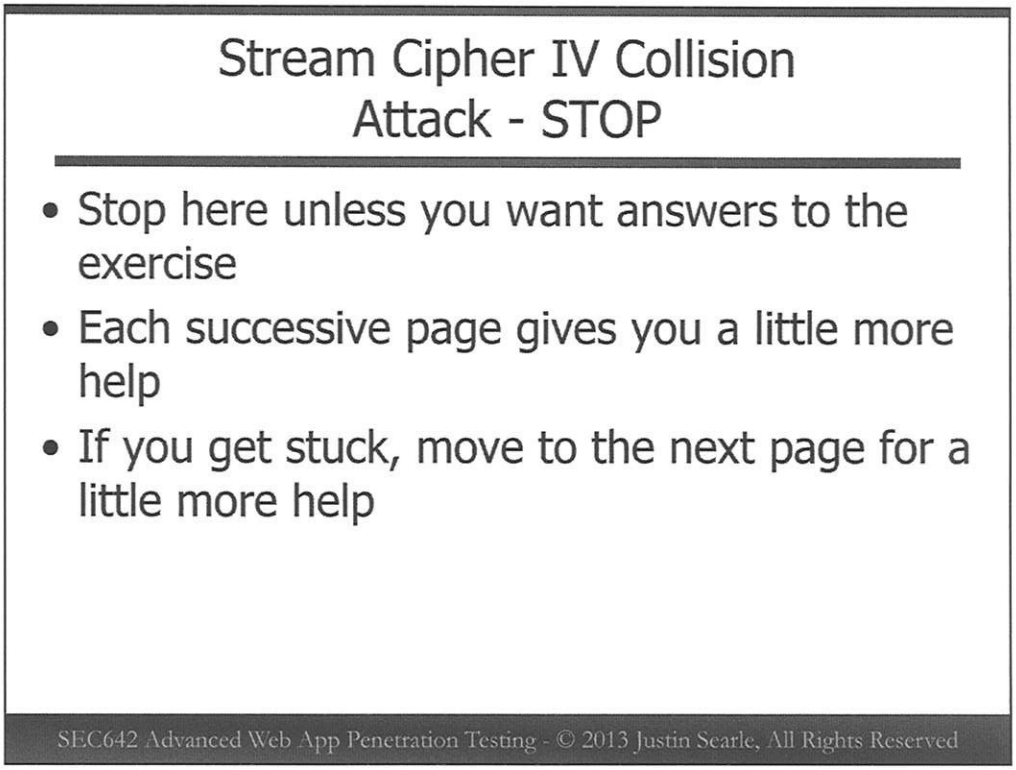
In this lab exercise you'll target a web application at http://fileboss.sec642.org. During your penetration test you discover that this internal application is used by employees to encrypt and store sensitive documents. A second application is used to decrypt files, but you don't have access to that system.

Your target is to retrieve and decrypt the file "message.txt". Be aware that you may also upload files to the system with your current privileges.

For this exercise you'll need some basic tools:

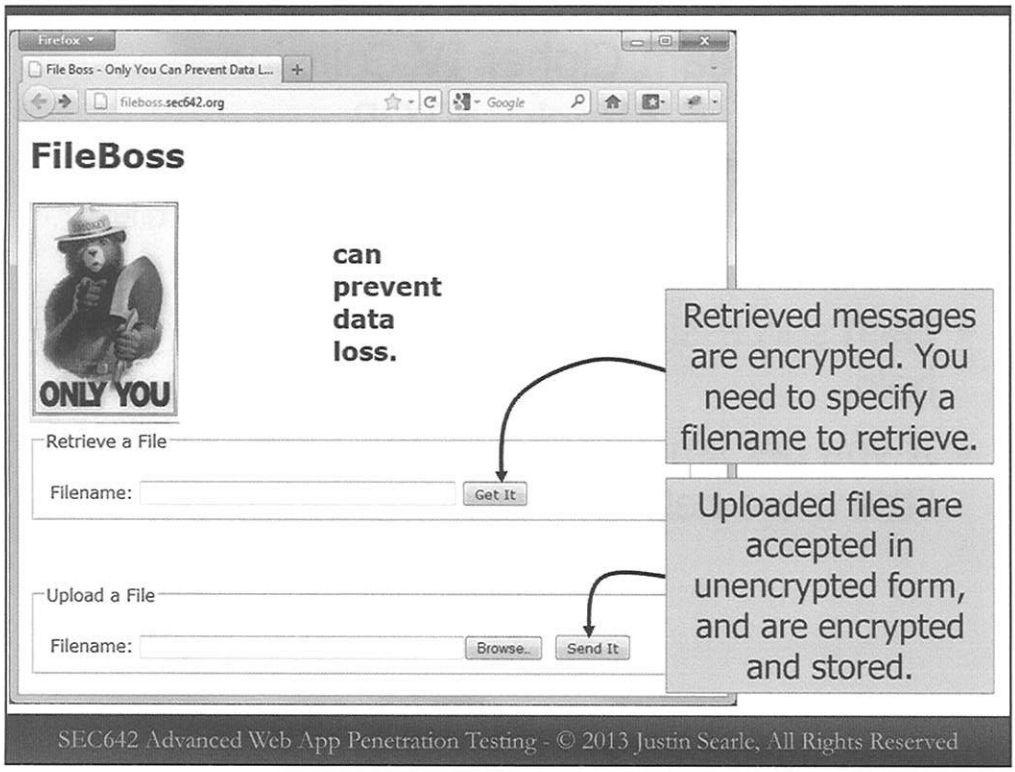
- Hexdump Tool: You'll need the ability to examine the contents of files in hexadecimal format. Windows users can retrieve a simple Windows hexdump tool from the URL specified on this slide. OS X and Samurai users can open a terminal prompt and use the xxd tool to obtain a hexdump of a named file.
- Calculator with XOR and hex representations: You will need access to a calculator that can XOR two hex values together. This feature is built-in to Windows and OS X calculators; Samurai users can use the tool "ghex2".
- ASCII lookup table: You will need access to an ASCII-to-hex lookup resource; from Samurai you can run "man ascii" or access an on-line resource for this manual page.

As an alternative to using these tools, you may wish to write a short script to decrypt the contents of the "message.txt" file. Two starter skeleton files in Python and Perl are available on the SEC642 file server at the URL's shown on this slide.



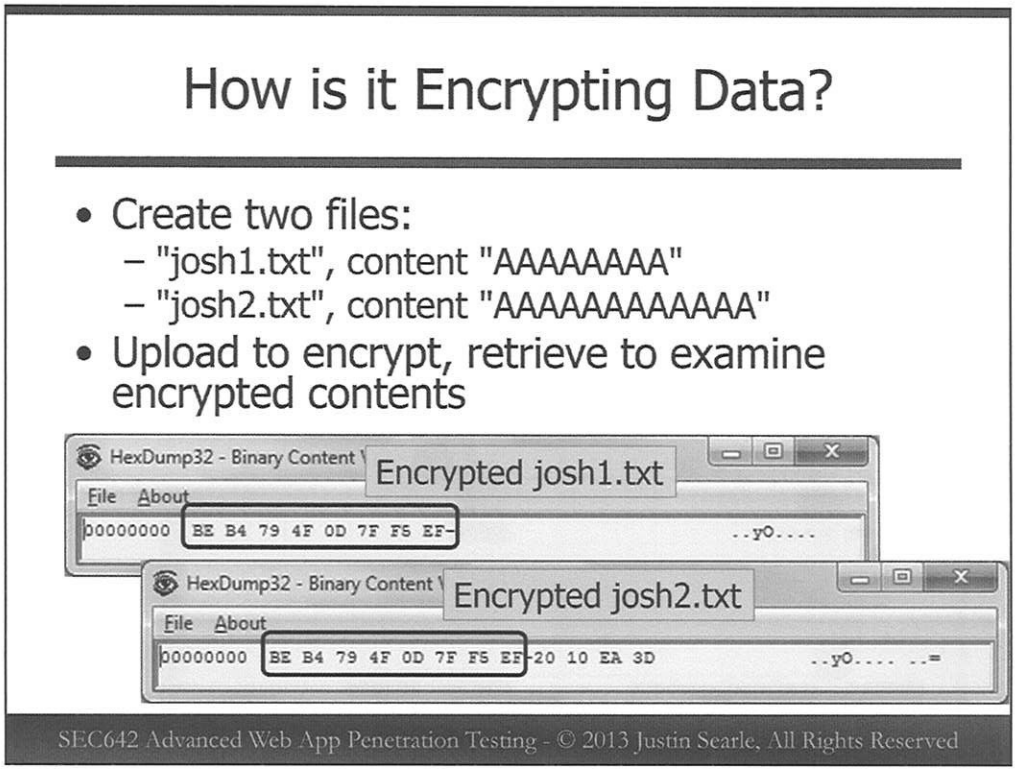
Stream Cipher IV Collision Attack – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



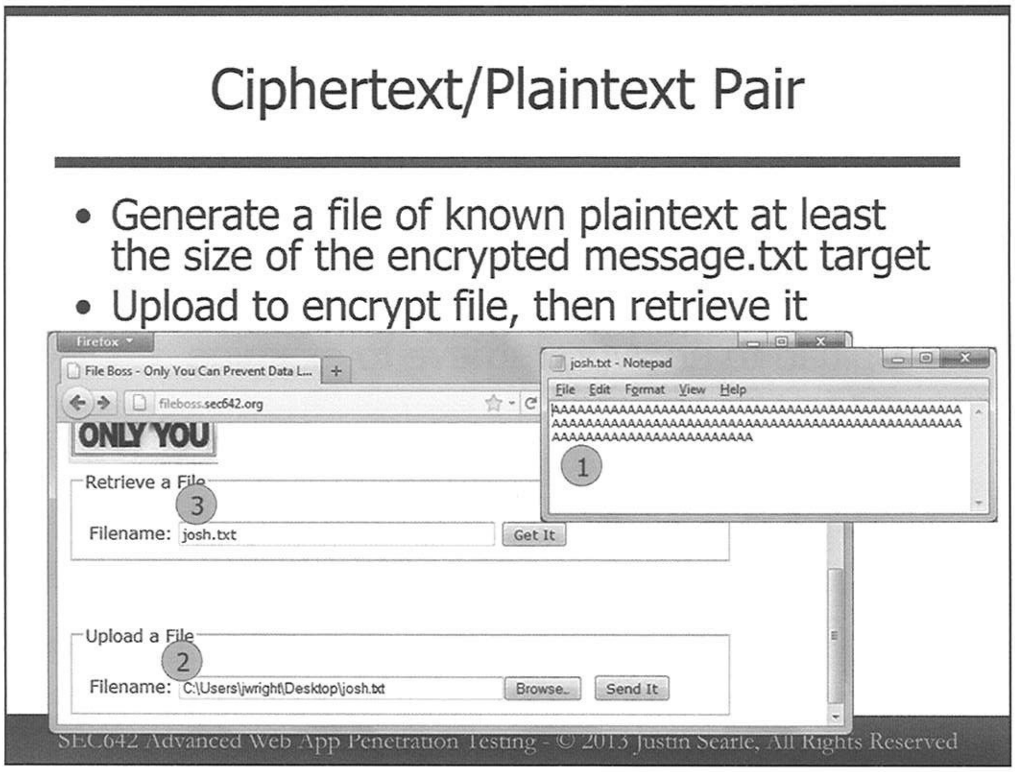
FileBoss

The FileBoss application is simple; you can retrieve encrypted files by specifying the filename and clicking "Get It". You can also upload files, which are encrypted and stored on the web server's file system.



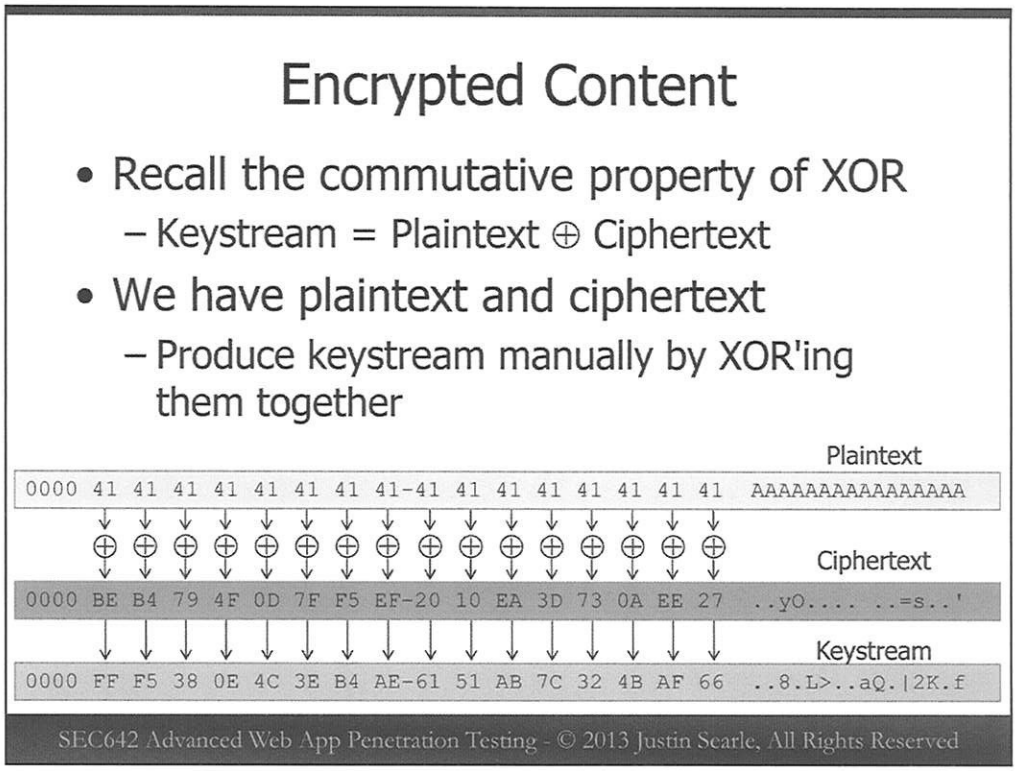
How is it Encrypting Data?

First we should determine how the system is encrypting data. Create two files with content similar to what is shown on this slide. Upload the files to encrypt them, then retrieve the encrypted counterpart. Looking at the encrypted files we see that the length is not incremented in a block size, and that the contents of the two encrypted values are identical up to the first varying byte of plaintext. This tells us that the system is using a stream cipher and that it is using an identical IV across multiple files.



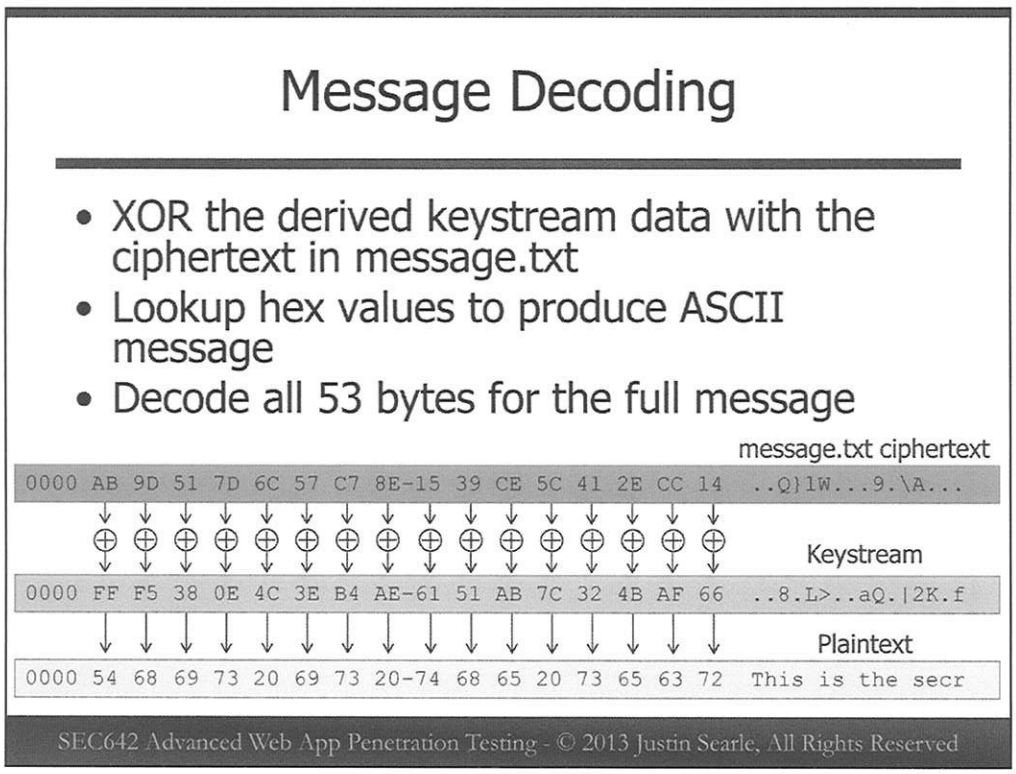
Ciphertext/Plaintext Pair

We want to recover keystream data to use for decrypting the "message.txt" file. Create a file similar to the one shown in this slide with a sufficiently large number of known plaintext characters. Upload the file to encrypt it, then retrieve the encrypted file.



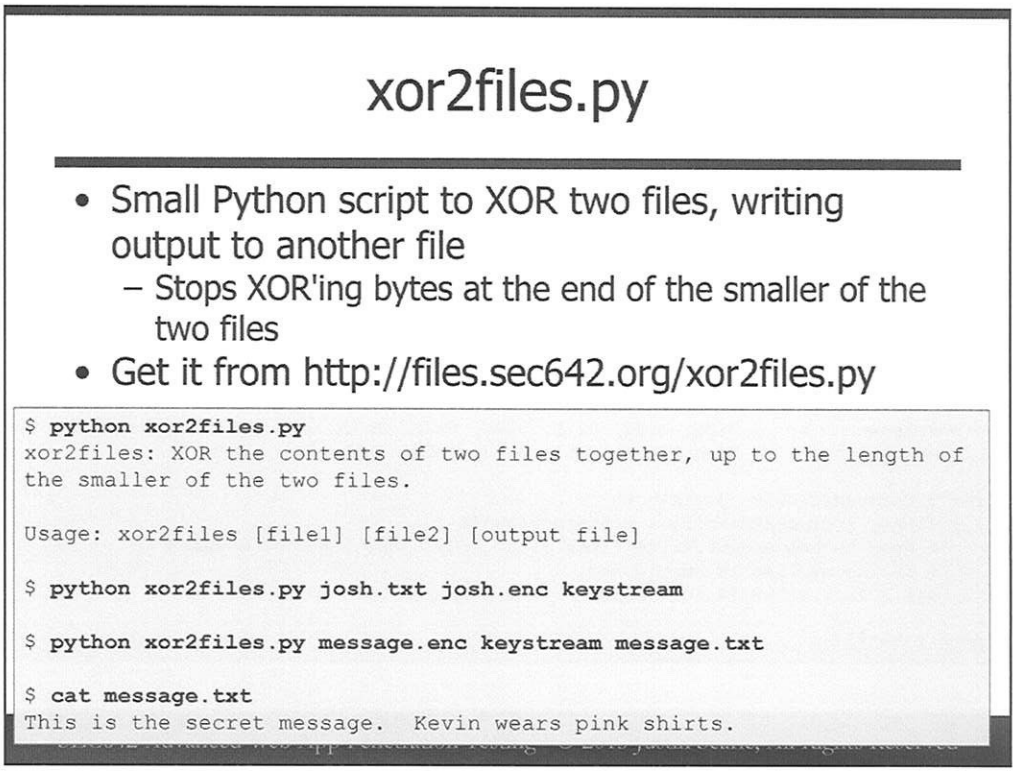
Encrypted Content

Recall the commutative XOR property that we discussed earlier in this module. We can recover keystream data by XOR'ing the encrypted file we generated with the plaintext content, as shown on this slide. If you aren't using a script, you will have to do this manually using a hex editor, hex calculator and pen and paper. After XOR'ing the plaintext and ciphertext together we recover the keystream data.



Message Decoding

With the keystream data we can recover the plaintext content of "message.txt". The first box shown on this slide is the content of the encrypted "message.txt"; the second box is the recovered keystream data. XOR'ing the two values together we start to see the beginning of an ASCII message "This is the secr". Continue to XOR the values together to recover the entire message up to the length of the encrypted "message.txt".



xor2files.py

To simplify the process or recovering keystream and decrypting content you can use the script at the URL shown on this slide. Simply run the script with the two files to be XOR'd together along with the name of an output file.

Here, we run xor2files.py twice; once to XOR the "josh.txt" plaintext with the "josh.enc" ciphertext to produce keystream data. Next we XOR the keystream data with "message.txt" to reveal the full secret message.

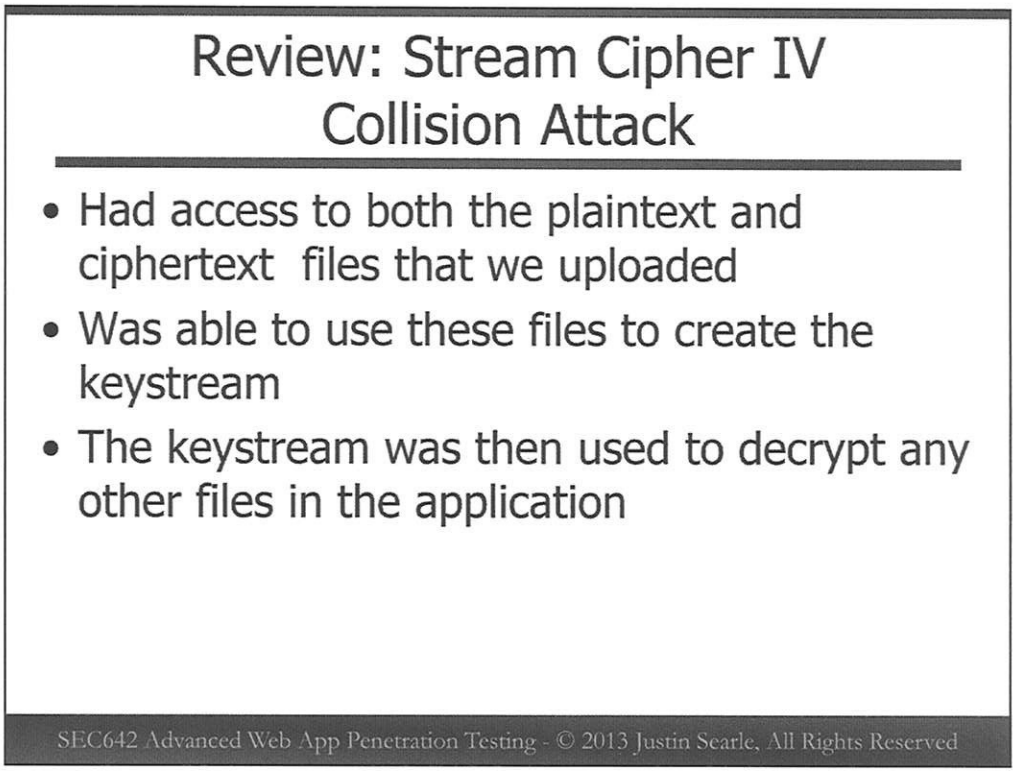
Script Solutions

- Python: http://files.sec642.org/filebossfail.py
- Perl: http://files.sec642.org/filebossfail.pl

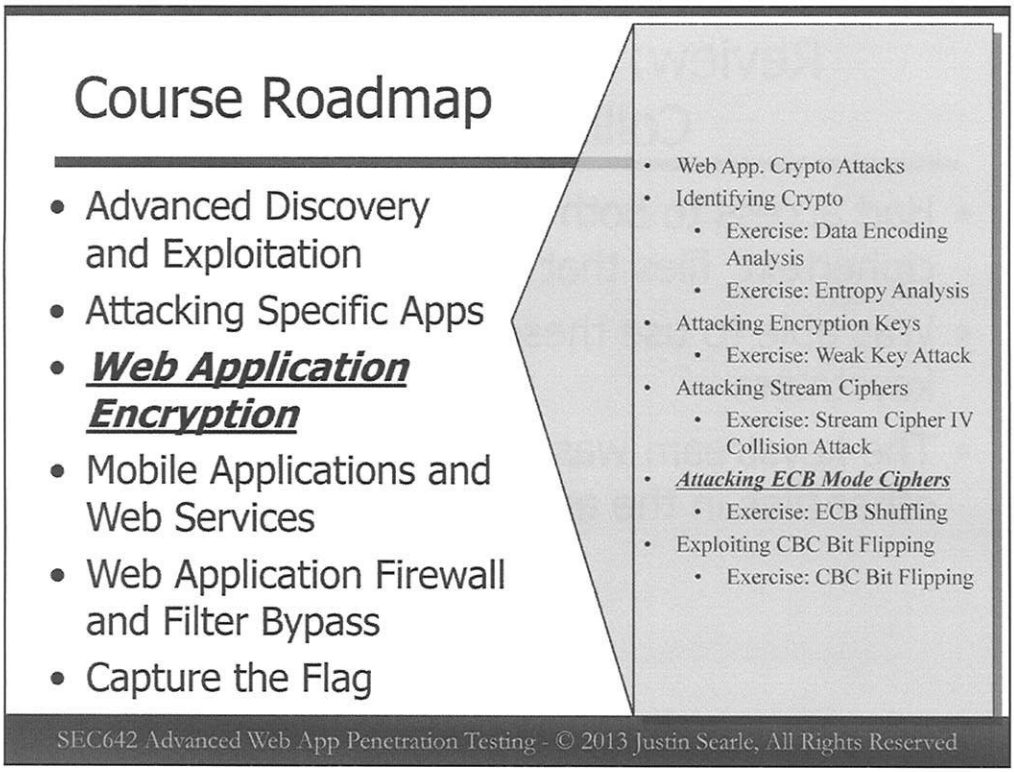
```
# This is only 8 bytes of each of the data sets; you want want to enter all
# 53 bytes of each data set to decrypt the entire message.
plainknown      =   ( 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41 )
cipherknown     =   ( 0xBE, 0xB4, 0x79, 0x4F, 0x0D, 0x7F, 0xF5, 0xEF )
cipherunknown   =   ( 0xab, 0x9d, 0x51, 0x7d, 0x6c, 0x57, 0xc7, 0x8e )

for i in xrange(0,len(plainknown)):
    cipxor = cipherknown[i] ^ cipherunknown[i];
    # Need to use stdout.write since print adds a new line, or a space
    # if the newline is suppressed.
    sys.stdout.write("%c" % (cipxor ^ plainknown[i]));

print("\n");
```


Script Solutions

Two script solutions are also supplied in Python and Perl at the URLs shown on this slide. The Python example on this slide only includes partial plaintext and ciphertext but can be expanded to decrypt the entire secret message.

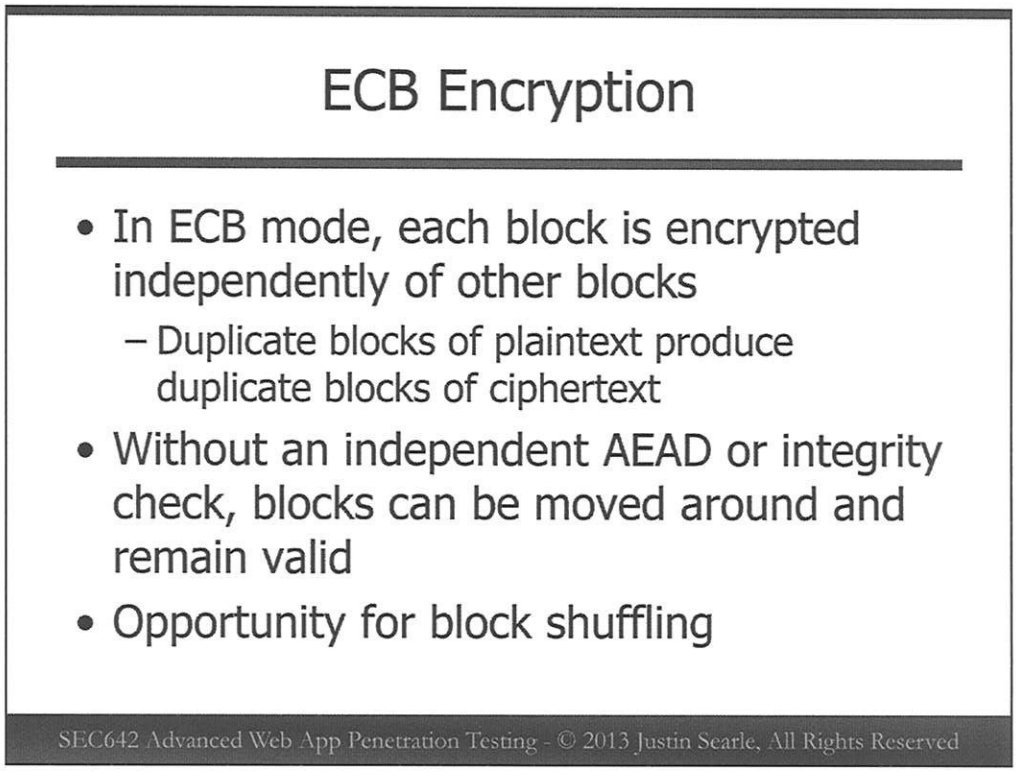


The key to this exercise is that we are able to create plaintext and encrypted files using the application, which then allowed us to recover the keystream used by the application. Because the application uses the same keystream to encrypt all files, we were then able to use the keystream to decrypt any other files on the system.



Course Roadmap

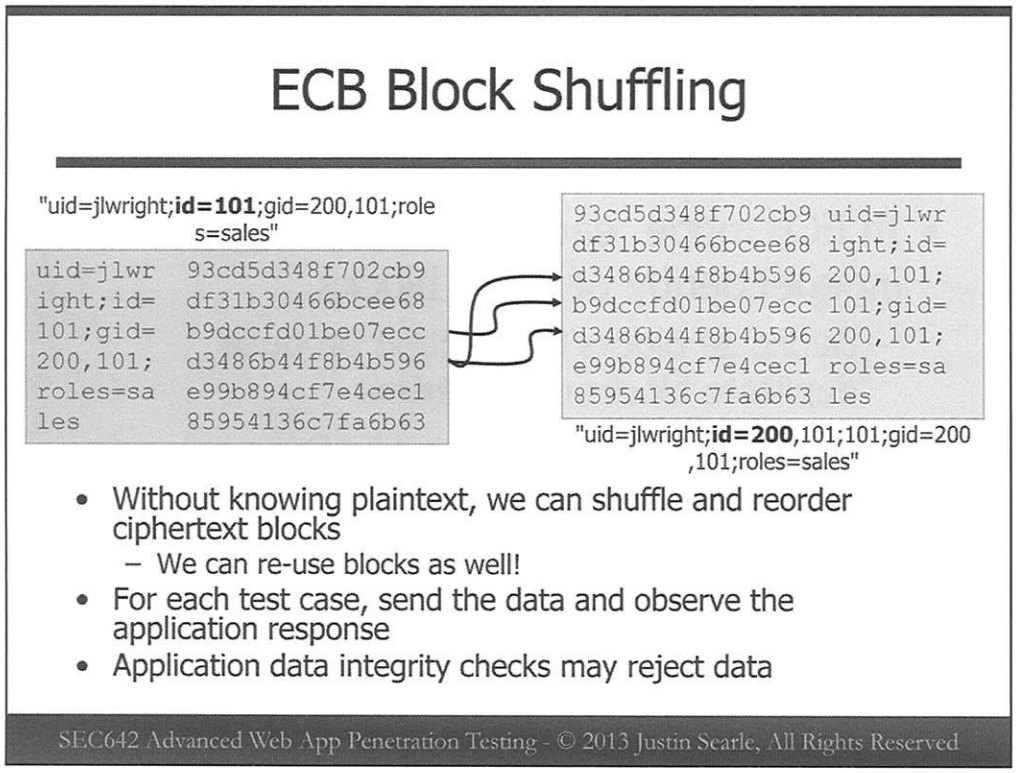
Next we'll look at exploiting web application systems using a block cipher in ECB mode.



ECB Encryption

When data is encrypted in ECB mode, each block is encrypted independently of other blocks. This process will lead to duplicate ciphertext blocks if there are duplicate plaintext blocks.

Aside from a data inference vulnerability where an attacker can identify duplicate plaintext blocks corresponding to the observed duplicate ciphertext blocks, ECB also suffers from a lack of integrity control that permits an attacker to reorder ECB blocks in the encrypted data set. This attack is known as ECB block shuffling, allowing us to manipulate the decrypted plaintext.

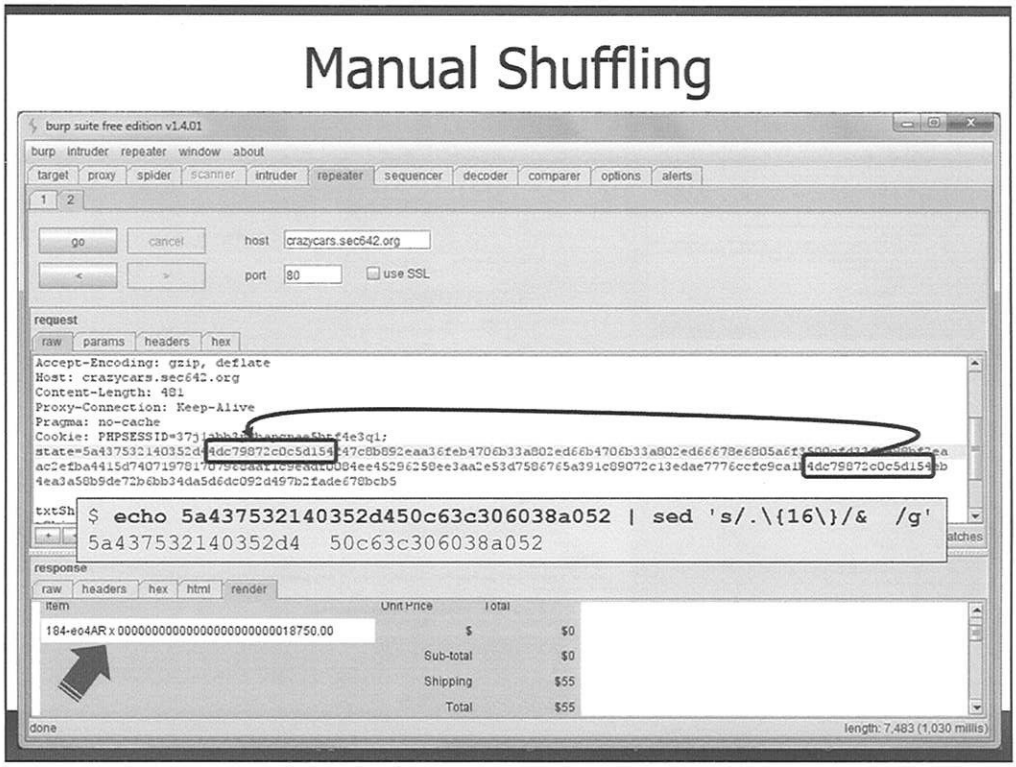


ECB Block Shuffling

In the example on this slide we show a legitimately encrypted string on the left, followed by the 8-byte encrypted blocks next to the plaintext. Note that the last block of plaintext, "les" is padded to make up the final block.

Since each block is encrypted independently, we can shuffle the block contents and still have the data decrypt to a reasonably valid value. In the example on the right we have replicated a single block, and moved some blocks around. Upon decryption, the plaintext string now reflects an "id" of 200, instead of the original 101, stealing from the block of encrypted content beginning with "200,101;".

When we attack a system with ECB block shuffling, we manipulate the position and possibly the repetition of the blocks, sending the shuffled blocks to the recipient. The recipient decrypts the data and may reject the content as invalid. (I In the example on this slide, the parser may reject the decrypted value of "id=200,101;" when it normally expects the numeric identifier to have a semi-colon immediately following). The attacker uses whatever response is available from the server to identify if the data was processed successfully after shuffling. If the shuffled data did not return a response that was desirable, we manipulate the encrypted blocks and try again.

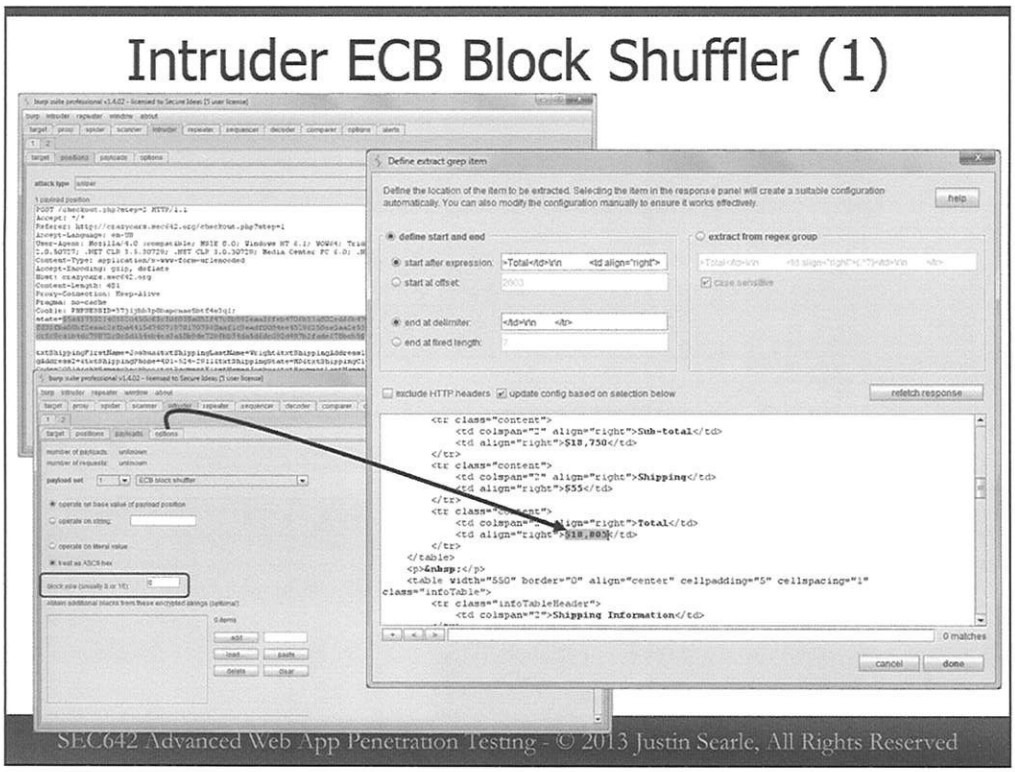


Manual Shuffling

You can manually shuffle ECB blocks using the Burp Repeater tool, as shown on this slide. We simply take the encrypted value and manually edit the order and repetition of the blocks to make the desired changes and send it to the target, observing any changes in the server's response.

In the case of this slide, the encrypted data is represented in hexadecimal ASCII values. In order to quickly shuffle the blocks without having to count each byte offset, we can add spaces after each block as shown with the echo and sed examples on this slide.

Manually shuffling blocks with Burp and Repeater or even a tool such as curl (which allows you to specify POST element data) is possible but unwieldy with large data sets. An automated tool that handles the shuffling in a systematic fashion will make the task much easier.

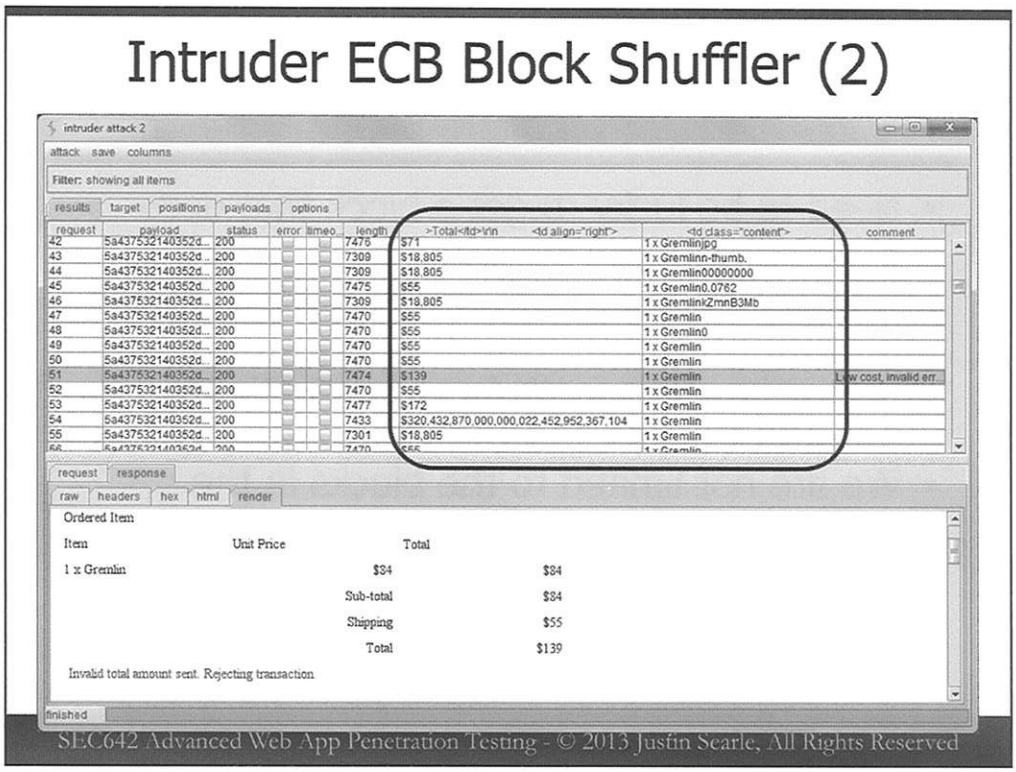


Intruder ECB Block Shuffler (1)

At the time of this writing, Burp Pro 1.5.04 includes an Intruder payload to shuffle ECB block data. On the Intruder tab, simply select the "ECB block shuffler" as the payload set. Burp will take the marked data and manipulate the blocks throughout the payload. For each block, Burp takes the block and replaces the original block with the new block.

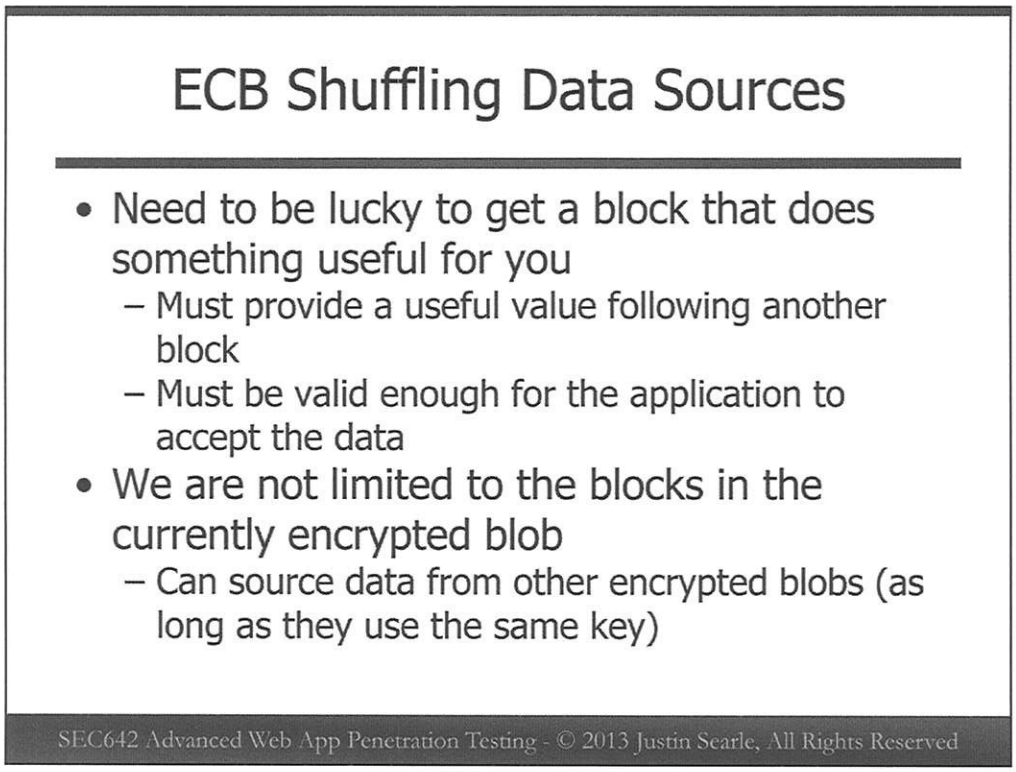
When performing ECB block shuffling you will get a variety of responses from the target system. The Intruder attack screen will show you the content of the manipulated ECB block data, the length of the response, the response code and other values, but this can still be difficult to sort through to determine if the ECB block shuffling created a useful condition.

In addition to showing generic response information resulting from the manipulated request, Burp can also display specific content taken from the result page. Before starting the attack, navigate to the Intruder options tab, then scroll down to the section marked "grep". Click the "extract" tab then click “new” to open the "Define extract grep item" window shown in this slide. Using Intruder's "grep" feature, we can highlight and mark specific values we want to see in the summary result information following the Intruder attack, such as the price of the item shown here. We can add several items as necessary to quickly view the results and determine if the Intruder attack generated a response that meets our attack requirements.



Intruder ECB Block Shuffler (2)

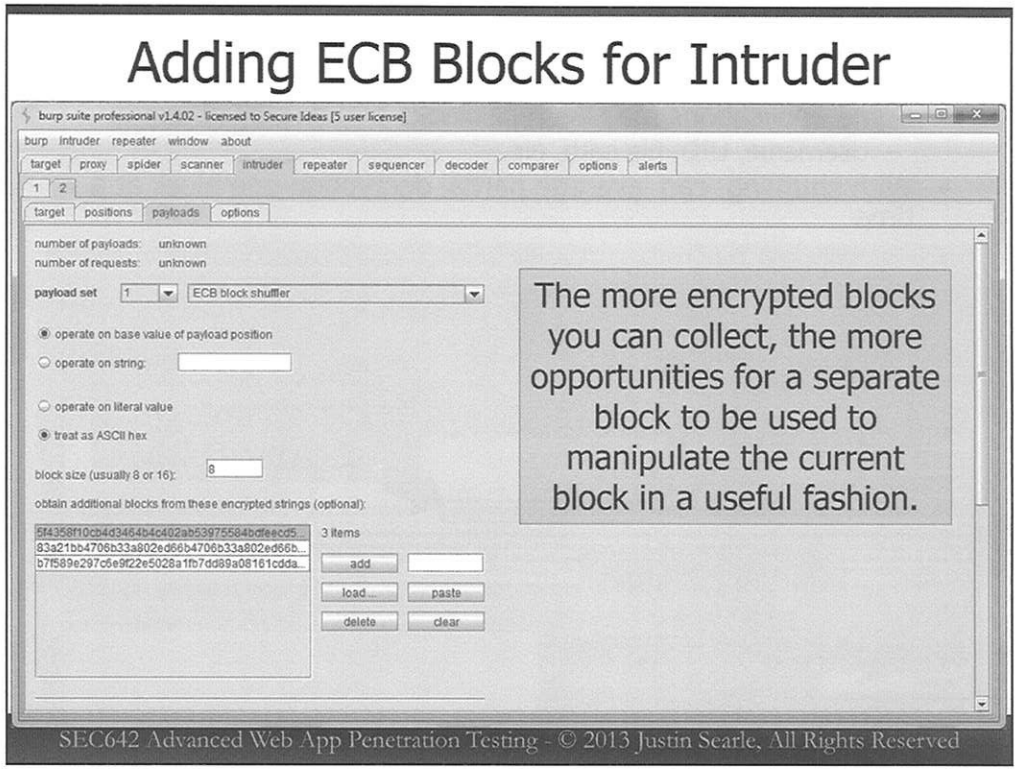
This slide shows the results from a Burp Intruder attack using the ECB block shuffler. For each shuffled block we get a one-line summary response displaying a sequential request identifier, the shuffled ECB block payload, the HTTP response code from the server, error and server timeout indicators, and the length of the response content. By using the Intruder "grep" feature we also add two new columns, showing the price in the total of the shopping cart and the name of the item in the shopping cart. These items can be arbitrary and selected by the penetration tester as needed.



ECB Shuffling Data Sources

In order to have ECB block shuffling be an effective attack, you'll need some luck to get a block that does something useful for you. The content of the block must "fit" where it is being added or upon replacing another block when decrypted so the application will process the data. Further, some blocks may generate a valid decrypted payload but may not provide a desirable attack result (such as replacing the price of an item in a shopping cart to a smaller value).

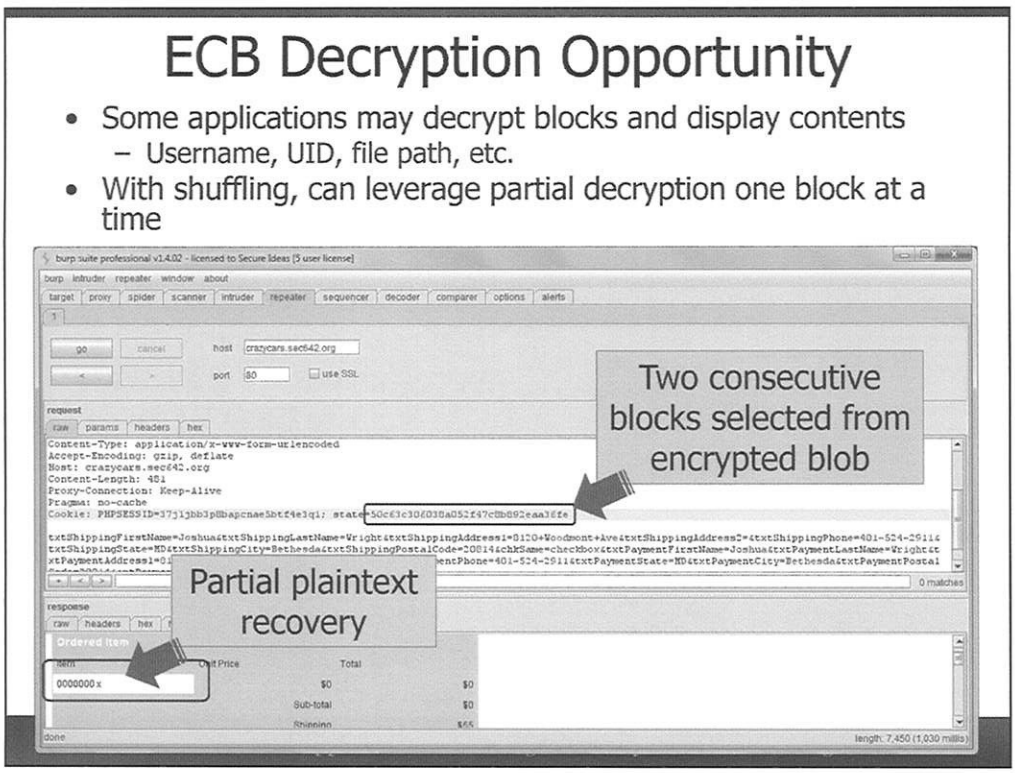
Fortunately, we are not limited to the currently encrypted blob of data for our ECB sources. If other data is also ECB encrypted on the site with the same key, we can retrieve those additional blocks and add them as additional source material for ECB shuffling. The more encrypted ECB block data we can collect, the more opportunity it creates for an encrypted block to decrypt to something useful in the shuffled data.



Adding ECB Blocks for Intruder

In Burp Intruder, navigate to the "payloads" tab and paste the additional encrypted blocks from the clipboard with the "paste" button. Alternatively, you can build a file of encoded block data which Burp can read from using the "load" feature.

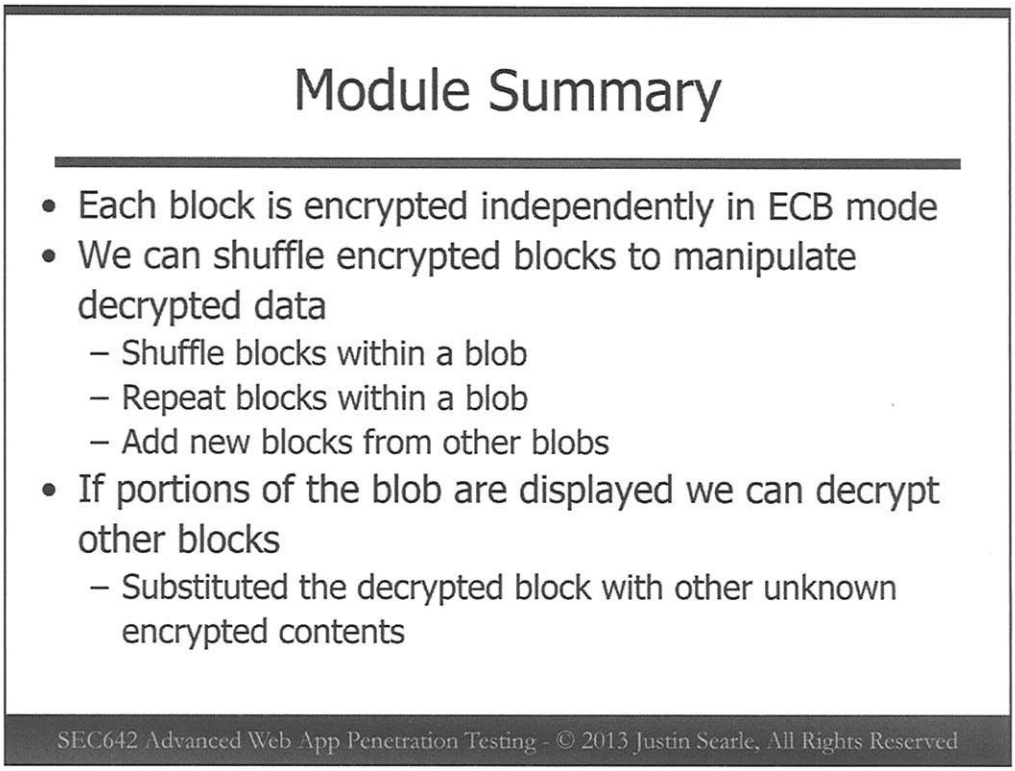
The process of collecting ECB block data varies from site to site, but consider the case of a shopping cart application. If you want to replace the price of an expensive item with a smaller price, you might collect ECB block data for each inexpensive item in the store, one at a time, and use the encrypted blocks for shuffling. Other unrelated content could also be beneficial, such as encrypted state tracking cookies or encrypted tokens following user authentication to the site.



ECB Decryption Opportunity

Using ECB block shuffling there is also a possible opportunity for the decryption of encrypted blocks. In the example on this slide, we have used Burp's Repeater tool to reduce the length of the "state" cookie to two consecutive ECB encrypted blocks. At the site's checkout page, we see a partial plaintext value in the column where the shopping cart item's name would otherwise be present, revealing the content of one block as "0000000x". We can replace the blocks used for the state cookie, iterating through all of the blocks from the original state cookie to decrypt the contents.

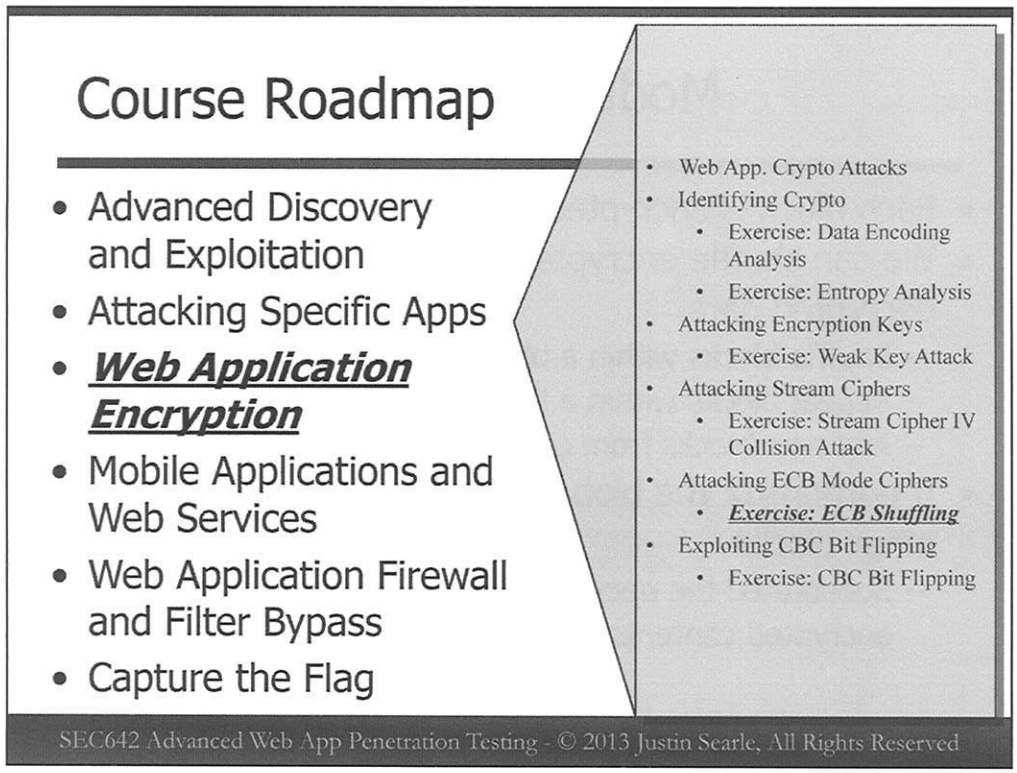
This ECB decryption opportunity is possible in sites where the application decrypts and displays a portion of the decrypted content in the server response. Keep an eye out for changing content being displayed following ECB block shuffling to identify ECB decryption opportunities.



Module Summary

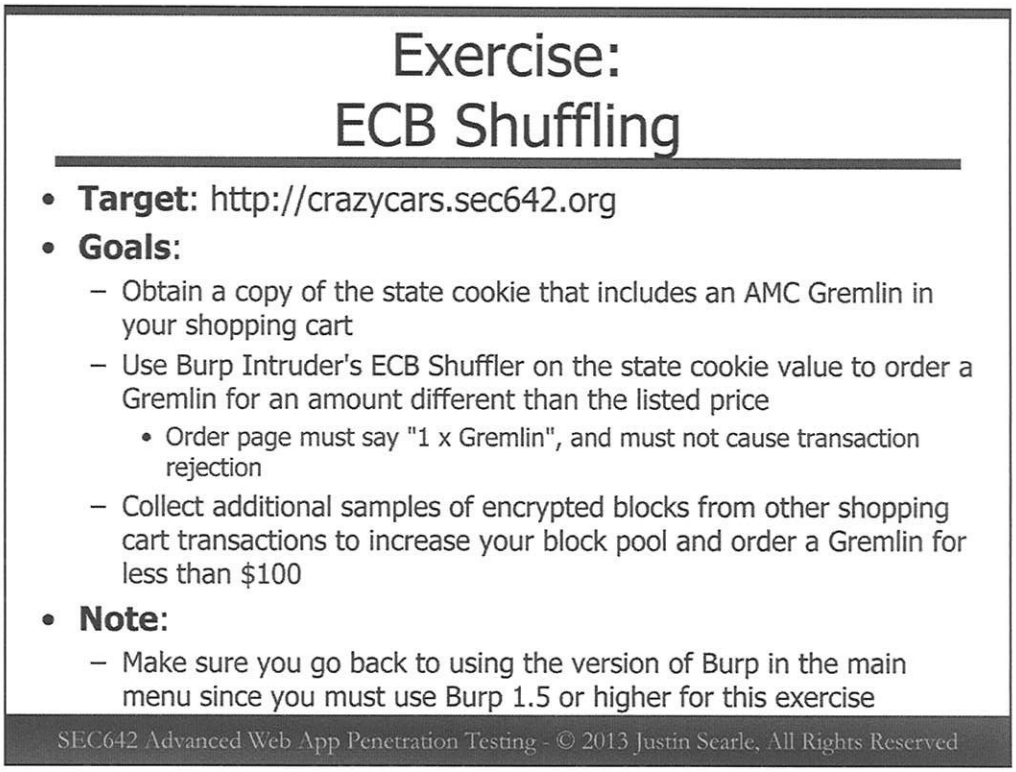
In this module we looked at an attack against ECB mode encrypted blocks. Since ECB mode independently encrypts each block of data, we have an opportunity to shuffle the contents of the encrypted data by repeating blocks, moving blocks around and adding new blocks from other encrypted data blobs. With luck and a large set of input blocks, we can manipulate the system in a desirable fashion. Although ECB block shuffling can be done with command-line tools such as curl or with Burp's Repeater function, the process is tedious and cumbersome. Automated tools such as Burp Pro's Intruder with ECB block shuffler make this attack much easier.

In some cases, the server may decrypt and display portions of the ECB block data (such as the logged-in user's name, or an item name in a shopping cart). We can leverage this opportunity to decrypt other ECB blocks and observe the decrypted response data by identifying the location of the block being displayed and swapping out the block for other encrypted blocks, recording the displayed, decrypted response.



Course Roadmap

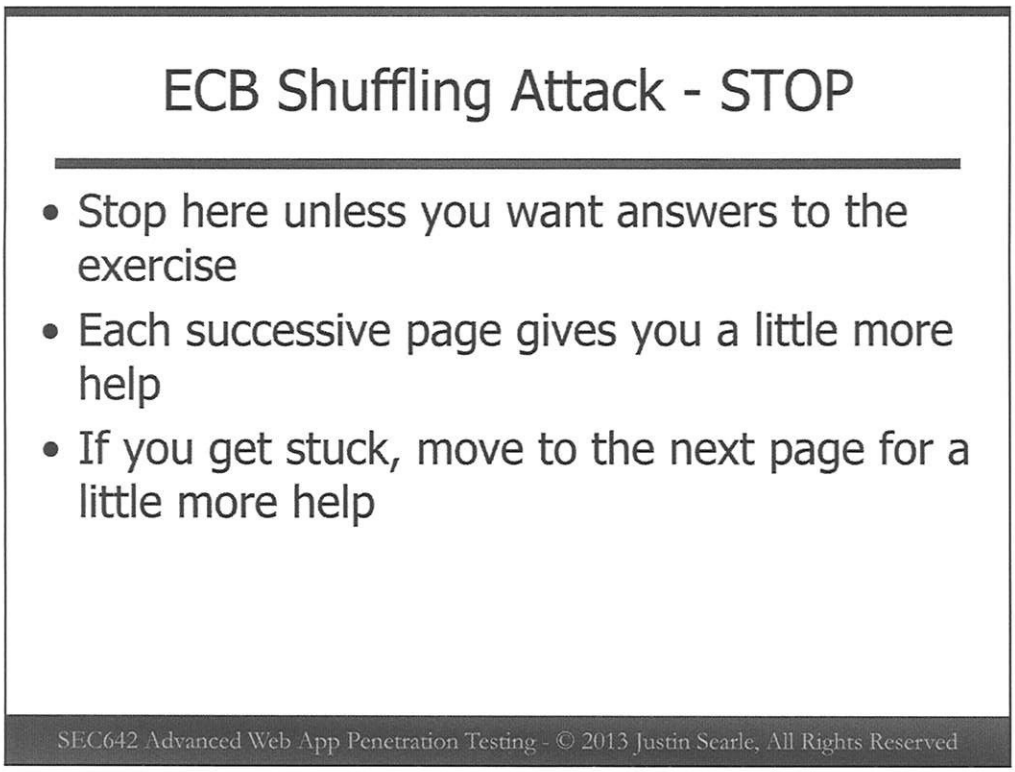
Next we'll look at exploiting web application systems using a block cipher in ECB mode.



Exercise: ECB Shuffling

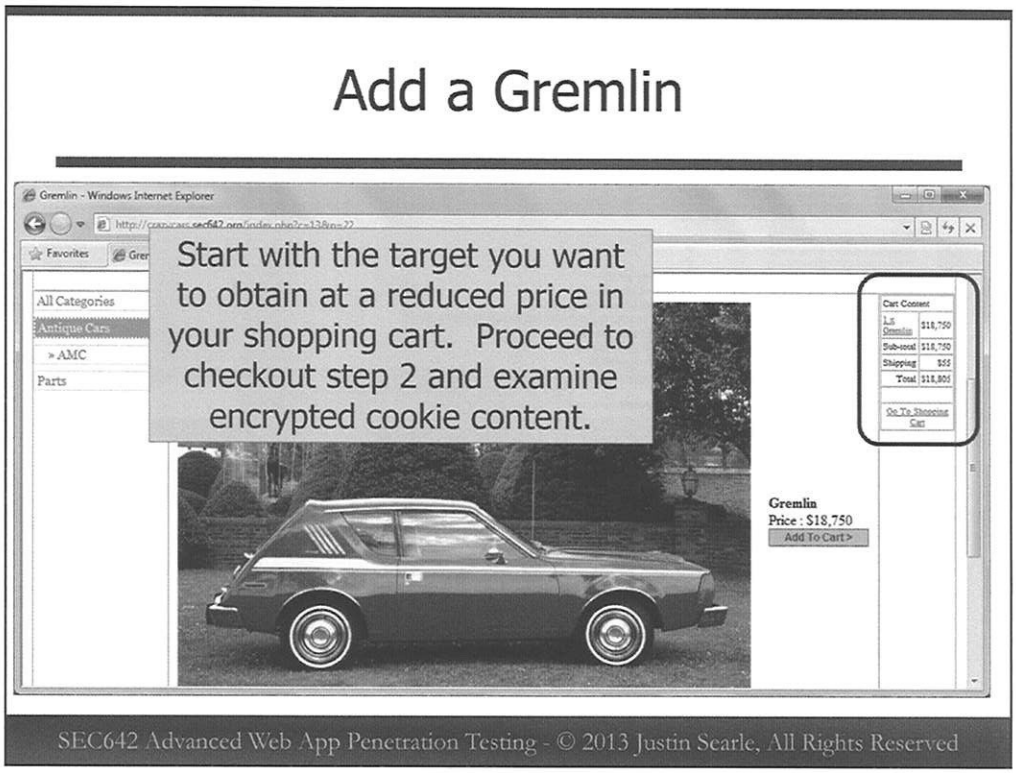
You will target the Crazy Cars site at http://crazycars.sec642.org for this exercise. You decide that you want to buy an AMC Gremlin, but that you do not want to pay their asking price for the car. In this exercise you'll use ECB shuffling to order a Gremlin for less than $100.

For success in this exercise, your order page must list an item "1 x Gremlin" and the transaction must not be rejected by system controls that detect invalid price manipulation attacks.



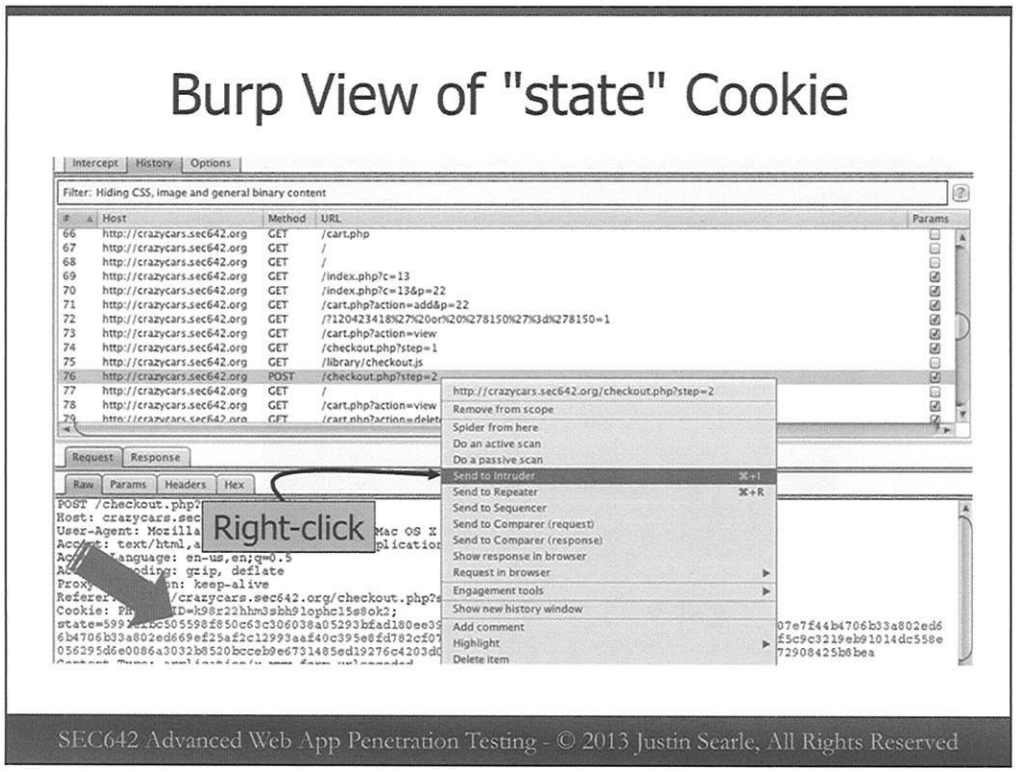
ECB Shuffling Attack – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



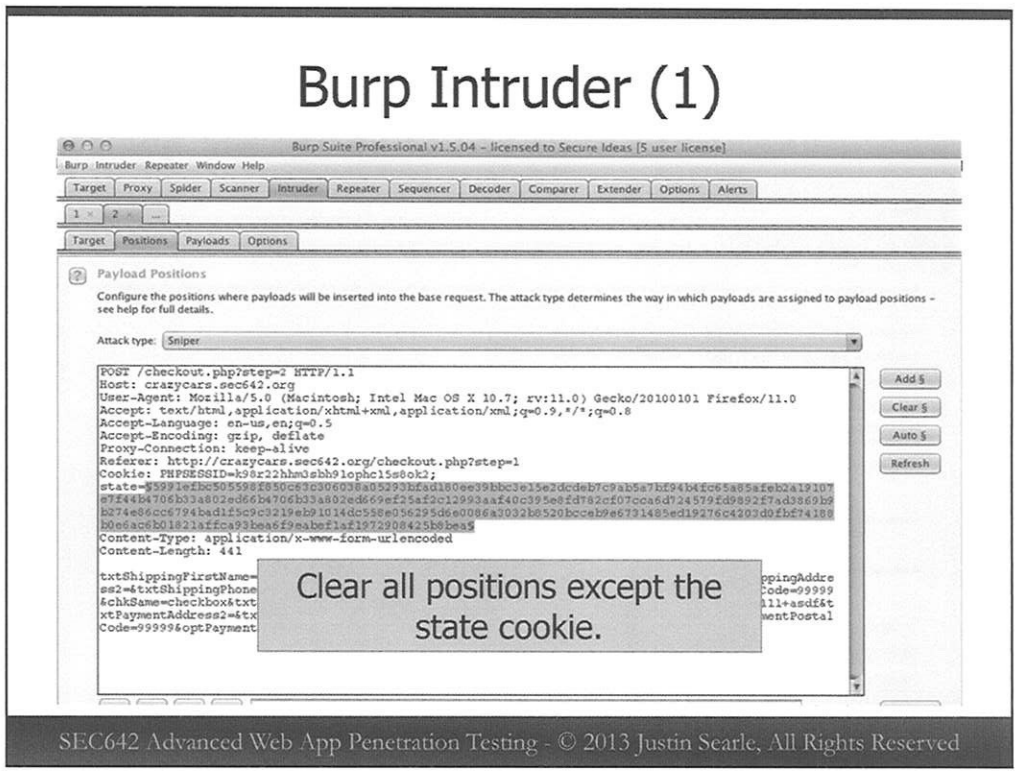
Add a Gremlin

For the attack, you'll want to start with a mostly-valid shopping cart. Add an AMC Gremlin to your shopping cart and proceed to the checkout page while capturing the transaction with Burp. Stop at step 2 of the checkout page (where you are asked to confirm your order).



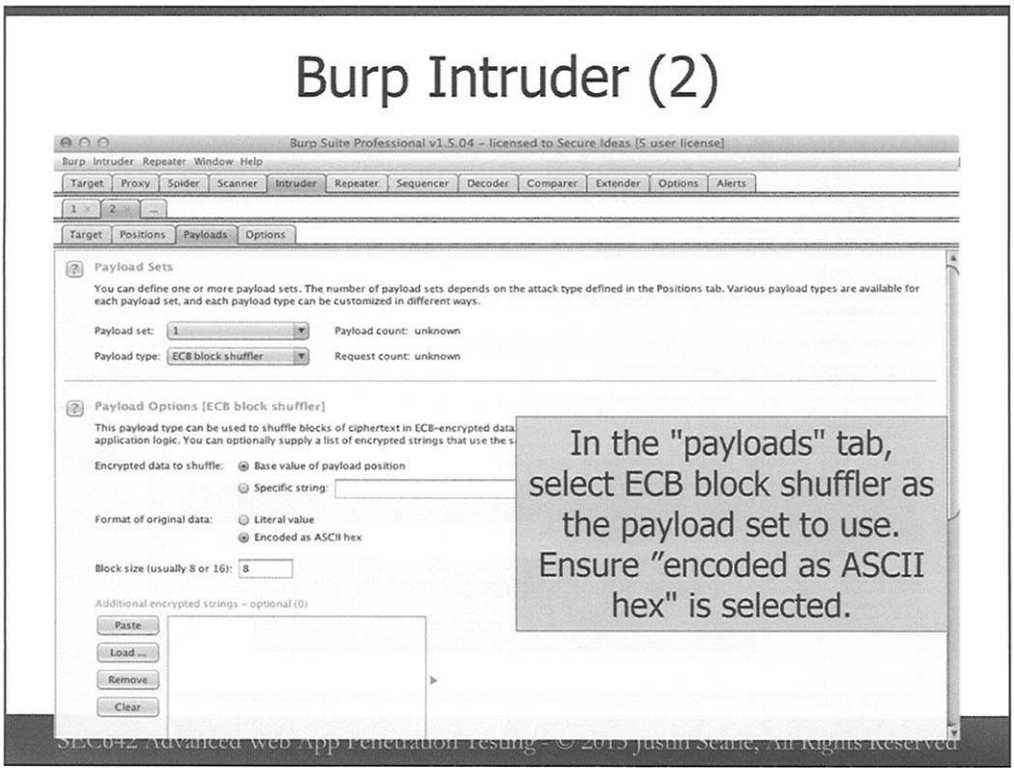
Burp View of "state" Cookie

Burp shows us the view of the "state" cookie in this slide. From our prior analysis in earlier lab exercises we determined that this cookie is ECB encrypted with an 8-byte block cipher (likely DES or 3DES). Right-click on this request and select "send to intruder" to leverage Burp's Intruder feature for ECB block shuffling.



Burp Intruder (1)

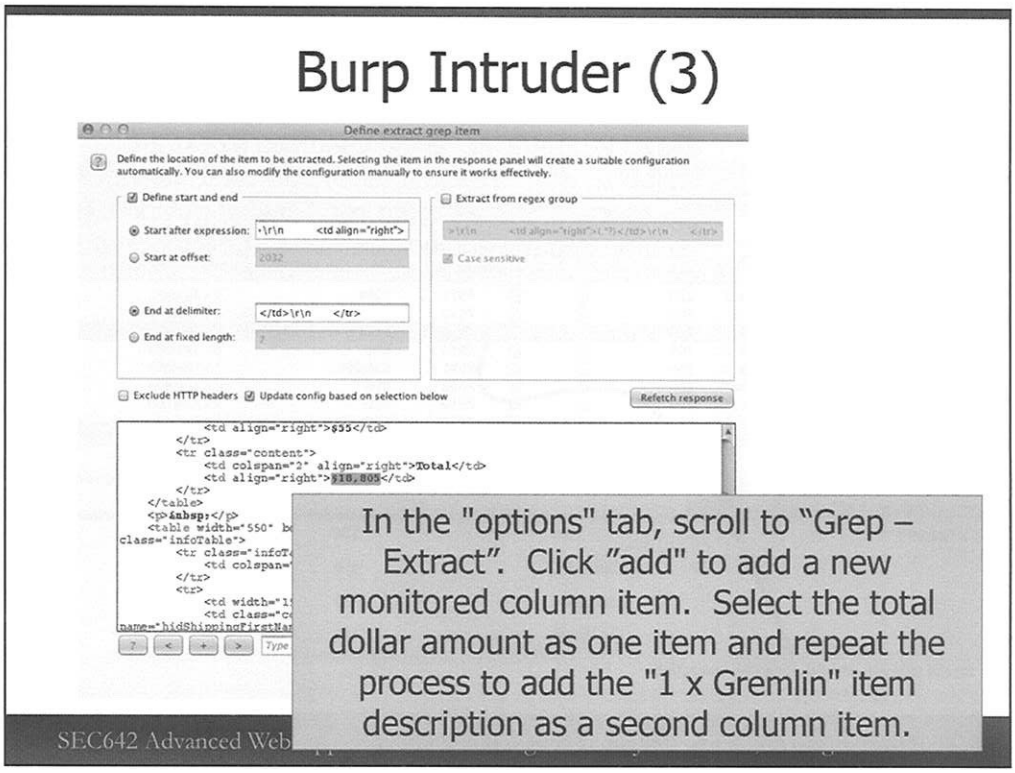
In Burp Intruder, click on the positions tab and clear all marked positions except for the state cookie content.



Burp Intruder (2)

In the payloads tab, select the ECB block shuffler as the payload set to use. If the ECB block shuffler option isn't available, ensure you have a version of Burp at least 1.4.02 or later by clicking the help | about menu.

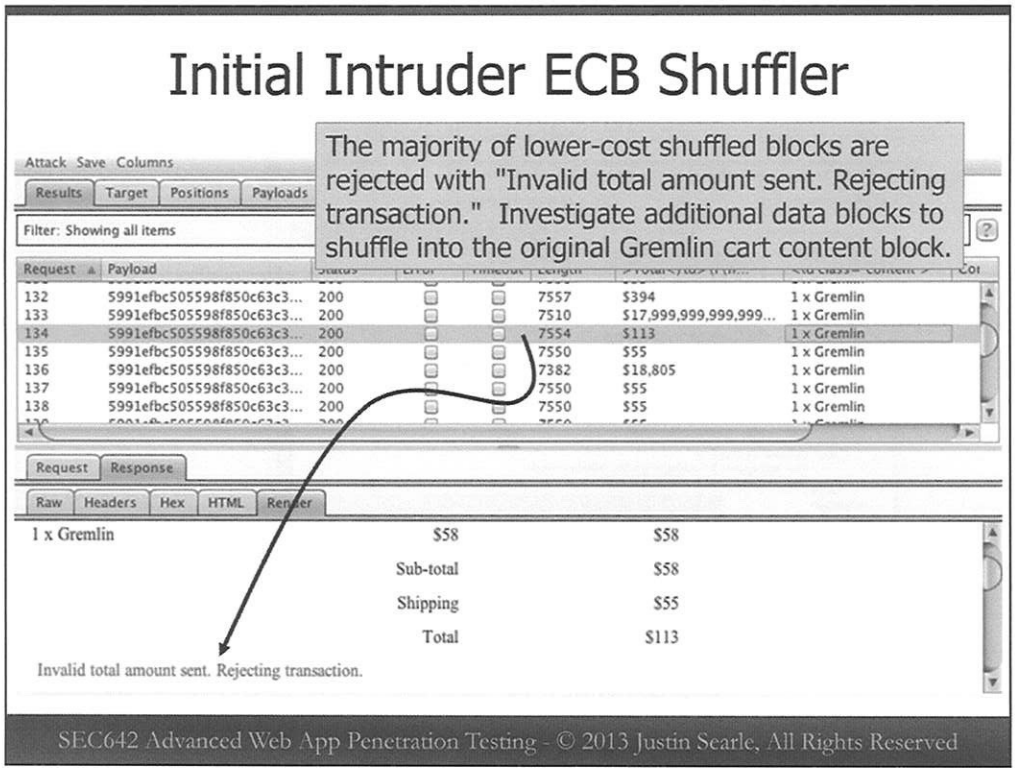
In the payloads tab, ensure the "encoded as ASCII hex" radio button is selected. Also ensure the "base value of payload position" is selected. Both are the default settings.



Burp Intruder (3)

In order to make it easier to review the results of Intruder's ECB block shuffling, navigate to the options tab and scroll to the "Grep - Extract" section. Click Add to add a new monitored column item. Add both the total dollar amount and the item description as new columns for display in the Intruder results window by highlighting the values in the HTML source. Click "done" to close the "Define extract grep item" dialog.

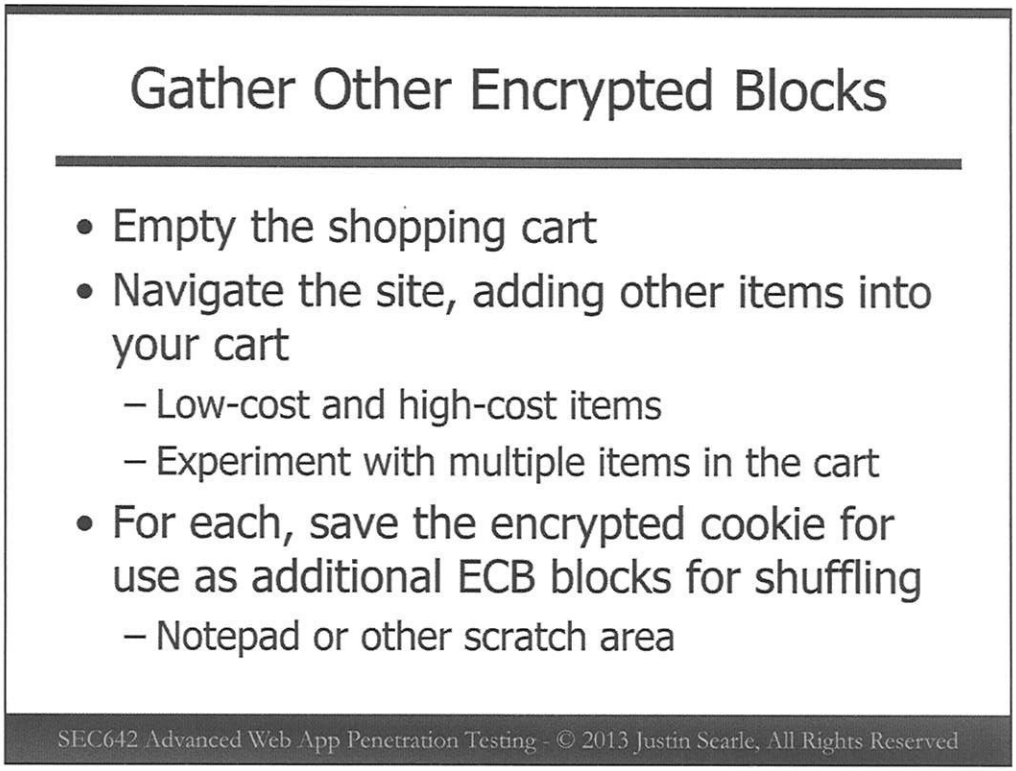
Once these Intruder settings are complete, click intruder | start to start the ECB shuffling attack.



Initial Intruder ECB Shuffler

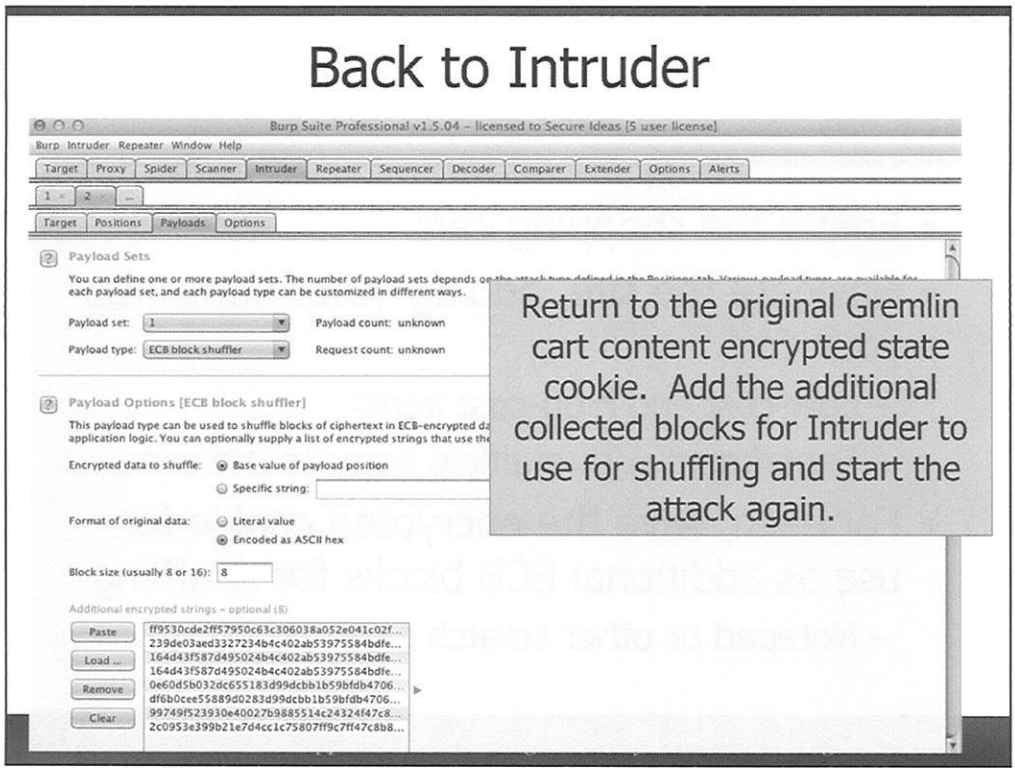
When shuffling just the contents of the Gremlin in the shopping cart you will likely see total dollar amount items that are small (such as the $182 example shown on this slide) with a valid item description column. When inspecting the response content however, you will see that the system product price validation mechanism rejects the price for the shopping cart item as shown on this slide. Further, the rendered page does not include the option to checkout since the validation mechanism triggered an exception.

Note, however, that the system does allow you to purchase a Gremlin for the paltry $17,999,999,999,999,999,689,420,177,408 price tag, shown. This does not meet our attack requirements however, so we need to investigate other options to bypass the price validation system.



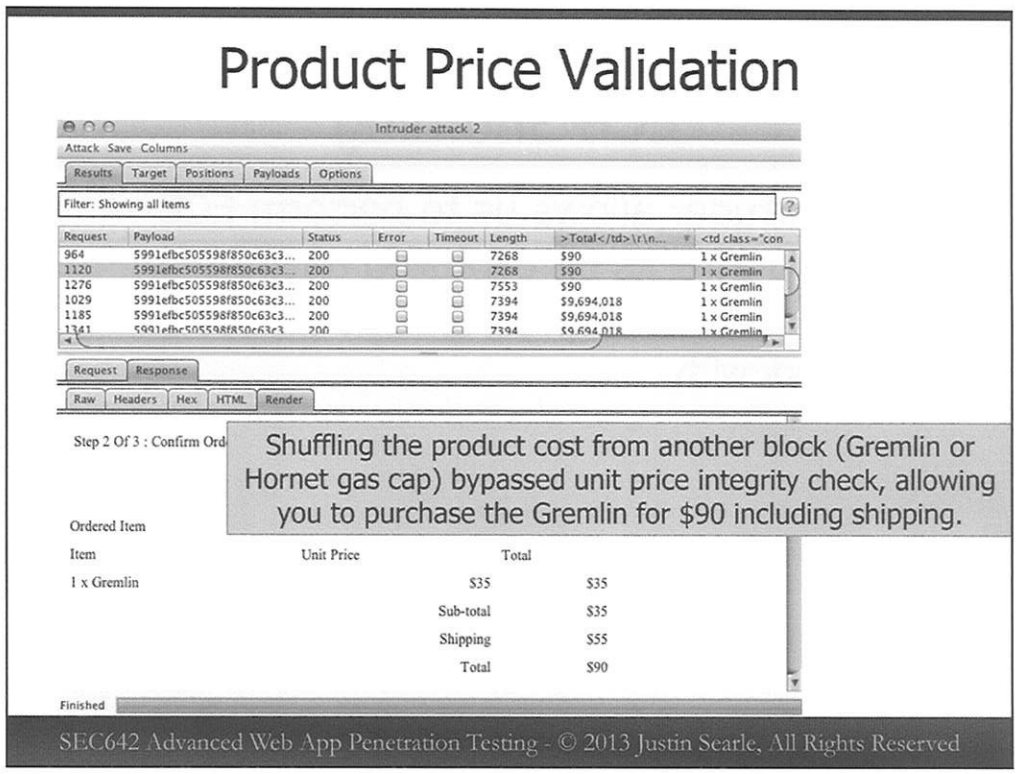
Gather Other Encrypted Blocks

Since the ECB encrypted blocks we shuffled in the Intruder attack did not give us a product price that bypassed the system price validation system, we need to search for other encrypted blocks that we can use as additional ECB block shuffling content. First, empty the shopping cart to clear out the previous "state" cookie content. Next, navigate the site, adding other items into the cart. Experiment with both high-cost and low-cost items, adding a single item to the cart or multiple items to the cart. For each experiment, use Burp to extract the newly encrypted ECB content and save it in a scratch area such as Notepad or other ASCII editor.



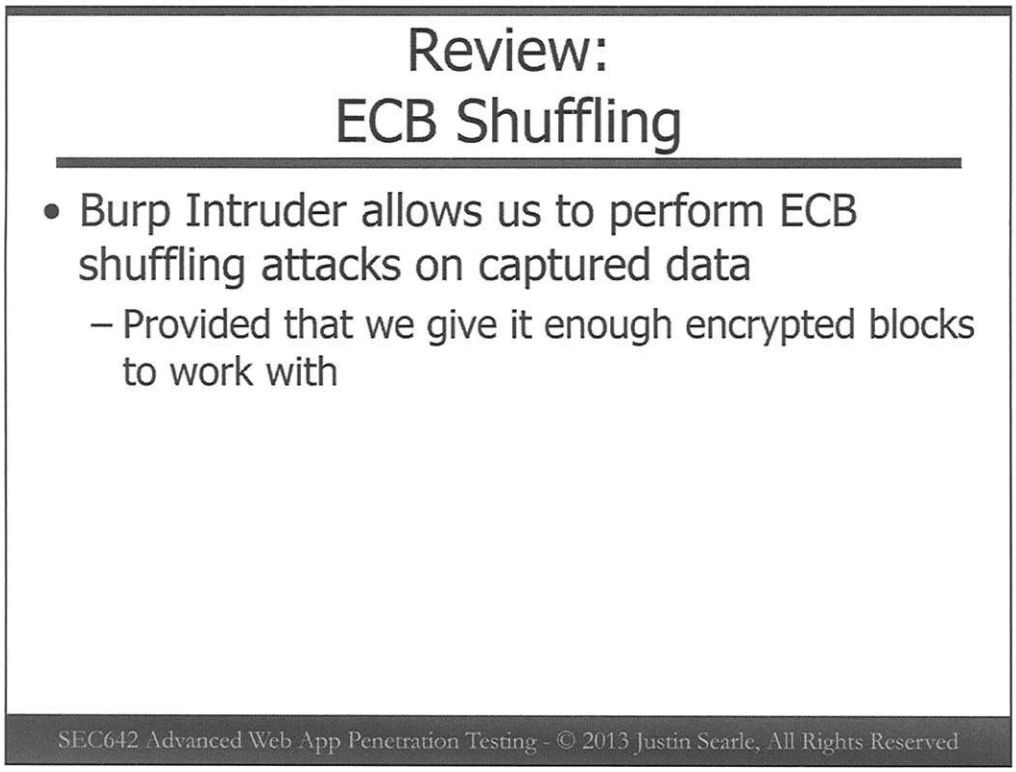
Back to Intruder

After collecting the additional ECB block content, copy the ECB blocks into the clipboard paste them into the original Intruder configuration for the Gremlin cart content. Ensure "base value of payload position" is still selected (it is the default, but has a tendency to switch to "operate as string" which is not used in our attack). Click intruder | start to start the attack again, this time leveraging the additional ECB blocks for shuffling.

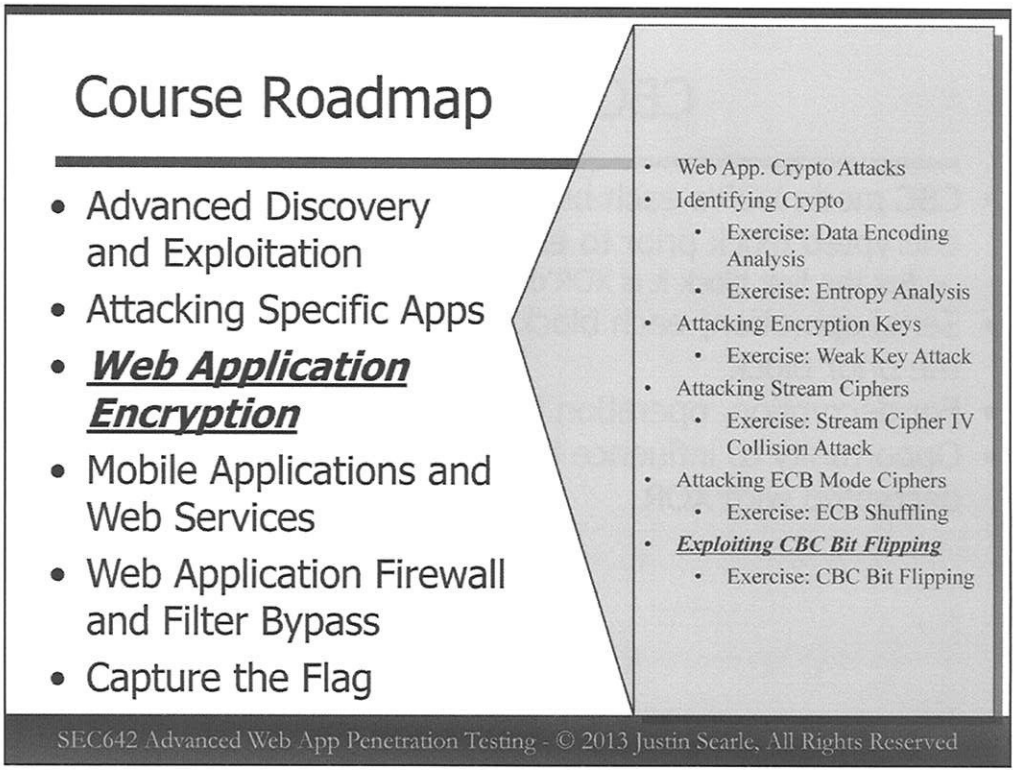


Product Price Validation

Examining the results from Intruder, we see several items for $90 with the correct item description. The ECB shuffler likely took the price of a gas cap and shuffled the price block into the place of the original Gremlin cost. The shopping cart system did not trigger an invalid price alter as the $35 price is a valid product price in the system, allowing you to purchase an AMC Gremlin for $90, including shipping. Aren't you lucky?

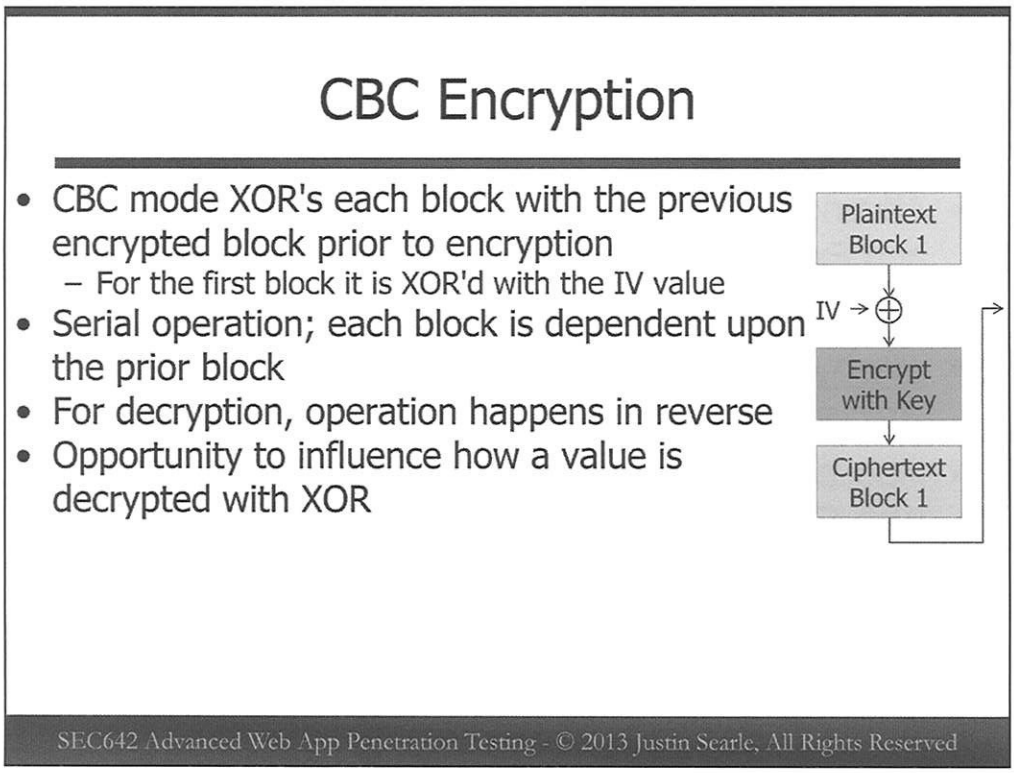


Burp Intruder allowed us to identify and manipulate the ECB blocks within the *state* cookie that related to the total price of our shopping cart. However, our initial testing did not produce a price that the application would accept. By collecting a larger sample of *state* cookie values, we were able to perform additional testing and find a value that the application would accept and that was under our $100 price limit.



Course Roadmap

Next we'll look at exploiting web application systems using a CBC bit flipping attack.

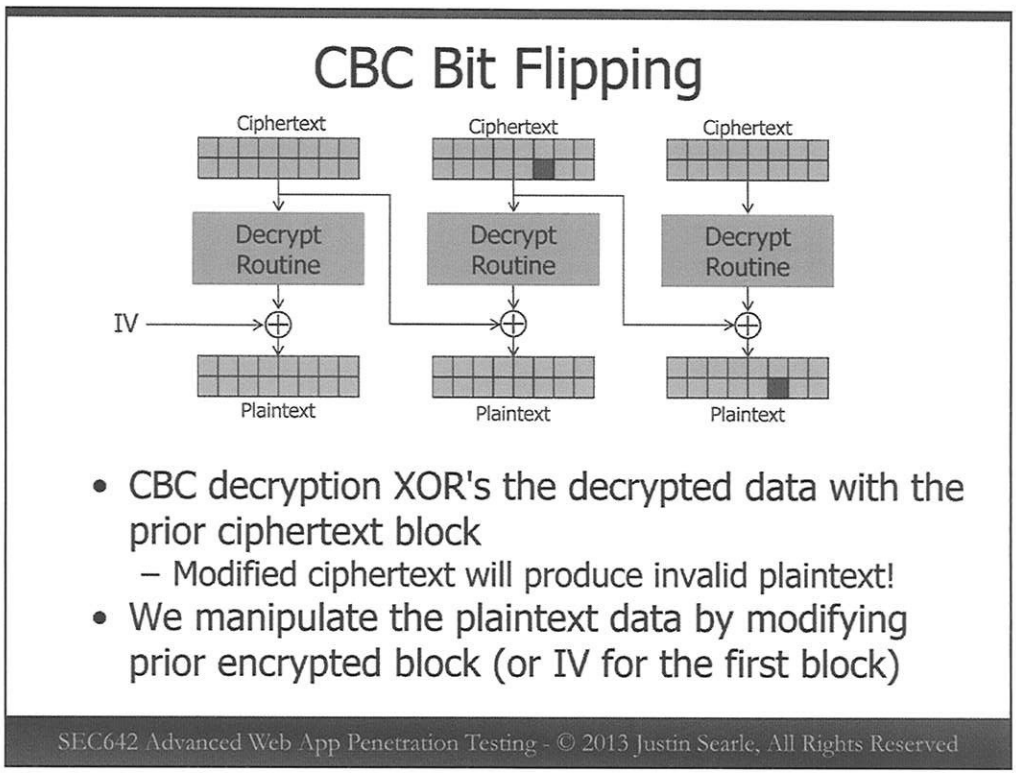


CBC Encryption

In this section we'll look at CBC bit flipping as an attack opportunity. Recall that when data is encrypted in CBC mode, each plaintext block is XOR'd with the prior ciphertext block before being encrypted with the key to form the next ciphertext block. The only exception is for the first encryption operation, where there is no prior encrypted block. In this case, the plaintext is XOR'd with the IV prior to being encrypted.

CBC encryption adds uniqueness to each encrypted block of data. Even if two identical plaintext values are encrypted with the same key, as long as the IV is unique then the ciphertext blocks will be unique. CBC encryption also defeats the block shuffling opportunity we saw in ECB mode encryption since the decryption of blocks is dependent on the block prior (or the IV for the first plaintext block). Shuffling CBC blocks would only decrypt to invalid and unpredictable content.

However, CBC does rely on XOR'ing the plaintext with the prior ciphertext, leading to an opportunity for us to predictably manipulate the content of decrypted blocks by modifying the prior ciphertext block (or the IV for the first plaintext block).

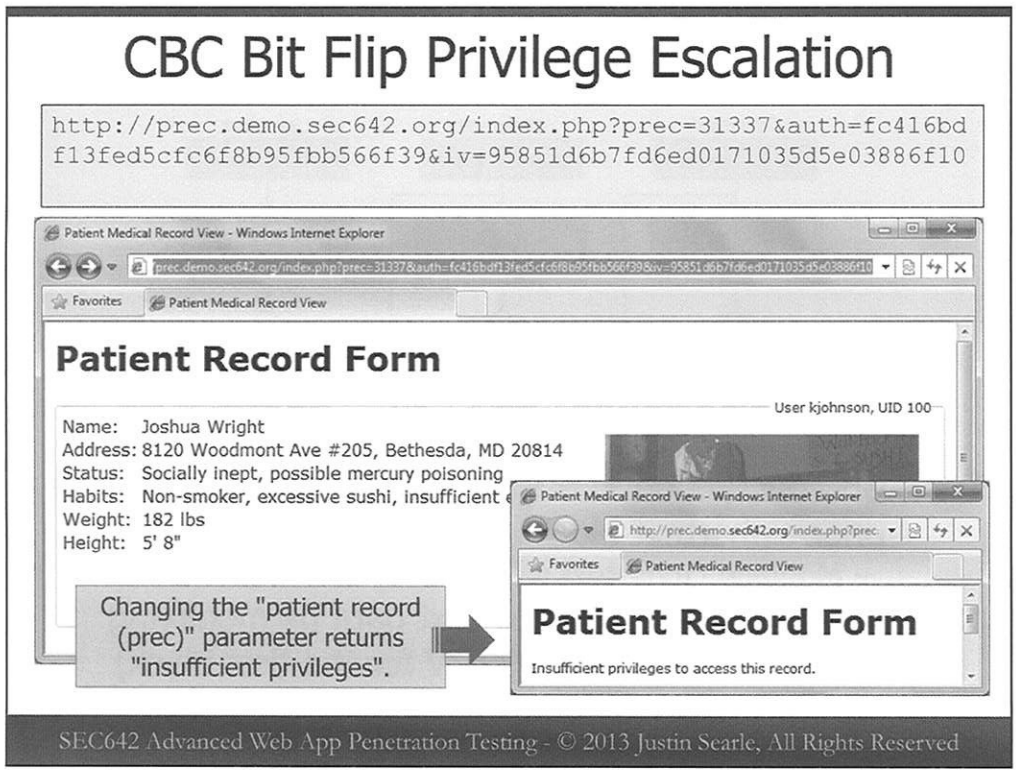


CBC Bit Flipping Attacks

While the previous slide showed the ECB encryption process, this slide shows the decryption process starting with the first block of ciphertext. Note that the ciphertext is decrypted and then XOR'd with the IV to produce the first plaintext block. The decrypted ciphertext is then used as the input of the next block, XOR'd 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, we manipulate the plaintext of the third block of plaintext (the right-most block shown). In order to manipulate this value, the prior ciphertext block is modified by flipping a bit. Once we flip the bit, we send the modified ciphertext result to the system and observe the server's response to our change. For example, we might attack an applications that performs a user ID check in an encrypted session variable, giving us an opportunity to manipulate the decryption process and gain escalated system privileges.



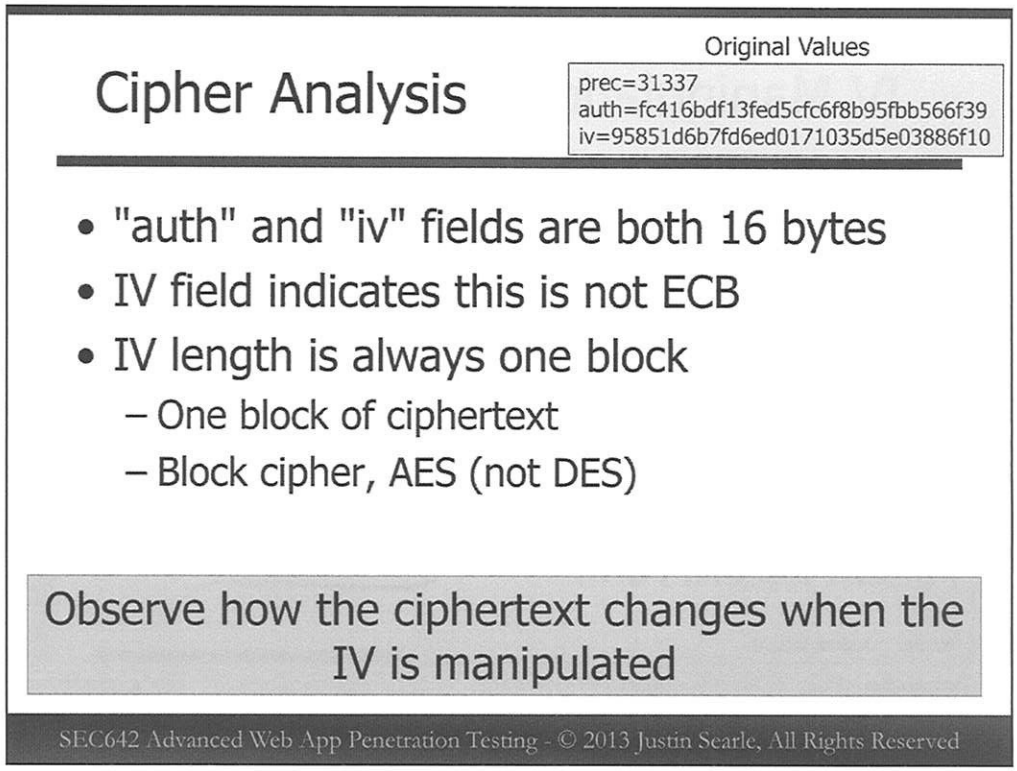
CBC Bit Flip Privilege Escalation

To demonstrate the attack opportunity present in CBC bit flipping, consider an application where authenticated session information is stored in a CBC encrypted form as shown on this slide, a mock medical patient record lookup application. Looking at the URL content, we see multiple GET parameters as follows:

- prec=31337 - By name, we can assume this parameter is the patient record number.
- auth=fc416bdf13fed5cfc6f8b95fbb566f39 - The auth parameter, presumably encrypted content identifying the user.
- iv=95851d6b7fd6ed0171035d5e03886f10 - The iv parameter is our IV for the encryption and decryption of the auth data.

The patient record form content shows information about the patient including the name, address, status, health habits, weight and height and a picture. Further, in the text box marking the patient data we see a smaller string identifier the user as "kjohnson" and a UID of 100.

Any attempts to change the patient record from "31337" to another value only returns an "Insufficient privileges to access this record" response.

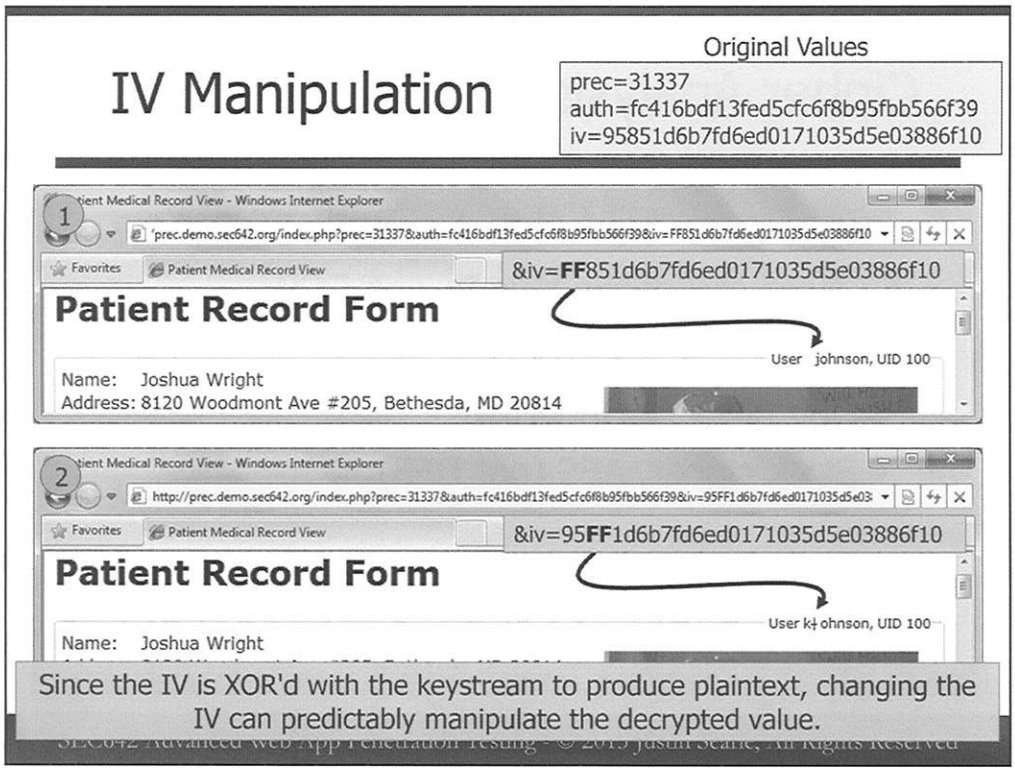


Cipher Analysis

Looking at the GET parameters, we see that the auth and iv fields are both 16 bytes. The presence of the IV indicates to us that the cipher is not ECB encrypted since ECB does not use an IV.

Looking at the auth data, we see the value is a hexadecimal encoding string 32 characters long, or 16 bytes decoded. This could be one encrypted AES block, or two encrypted DES blocks. However, since the IV is always the same length as the block length, and the IV is 16 bytes after decoding, we know the block length of 16 bytes. This tells us that the data is likely AES encrypted.

Upon noting an implementation like this we would start to manually manipulate the IV value and observe any changes in the application that result. Here, the IV is clearly marked as a separate parameter, however, many systems will embed the IV as the first block of the ciphertext value. Keep that in mind when evaluating the encrypted content, knowing that the IV may just be a portion of the encrypted data.

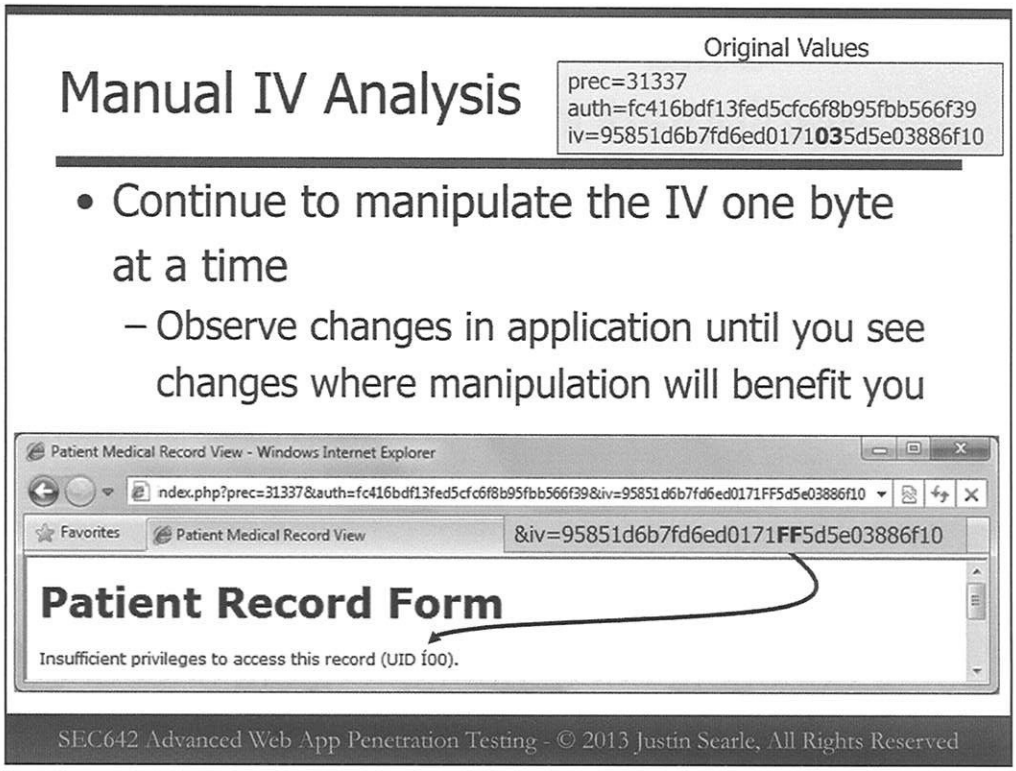


IV Manipulation

The preference of this author, when manipulating an IV, is to set all bits high (e.g. 0xFF) one byte at a time, observing the response from the server after each. This isn't absolutely required, through it is a good idea to limit your testing to one byte at a time (or even one bit or nibble at a time) to keep track of the bit changes that cause a varying response on the server.

In the first example on this slide we change the first byte of the IV to 0xFF from the original 0x95. When we see the server response, we see the username information that was previously "kjohnson" has now changed to " johnson". This reveals to us that the username information is the beginning of the ciphertext since that was the byte that was affected by changing the first byte of the IV. We can confirm this by returning the IV's first byte to 0x95 and changing the 2nd byte of the IV to 0xFF. This change causes the 2nd byte of the username to change, as predicted.

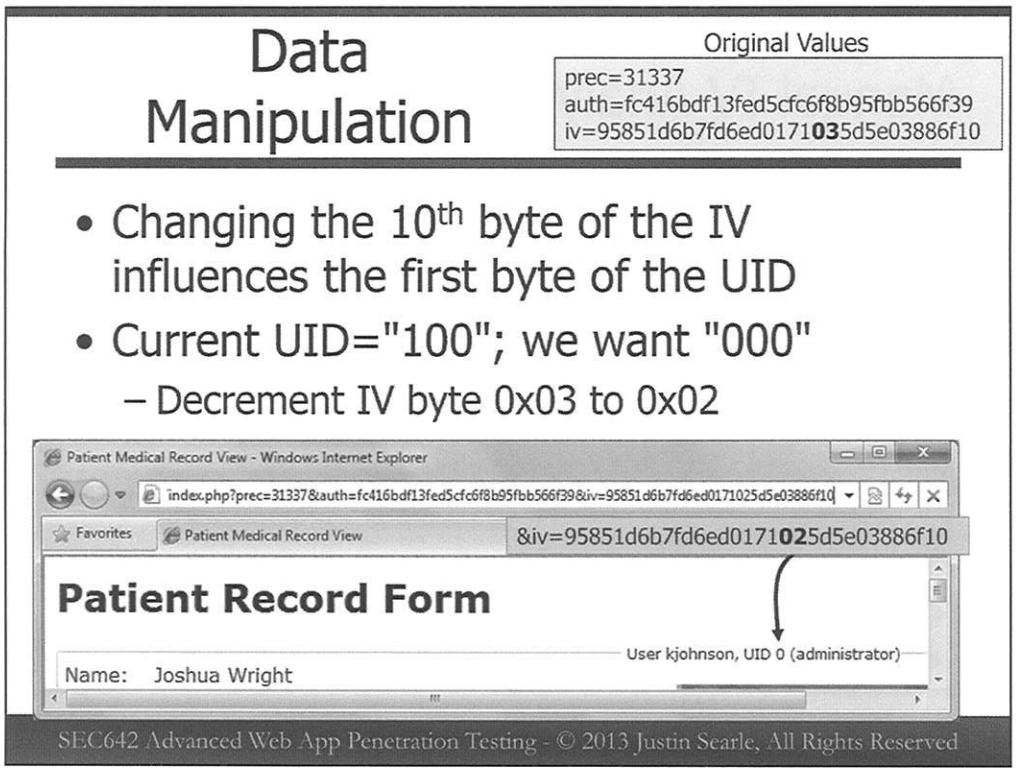
Since the IV is XOR'd with the keystream data to produce the plaintext, we can change the IV in a predictable manner to influence other bytes of decrypted plaintext.



Manual IV Analysis

Next we continue to manipulate the IV, one byte at a time, observing how it changes the server's response. Here, changing the 10th byte of the IV causes the UID value to change, revealing to us its position in the ciphertext block.

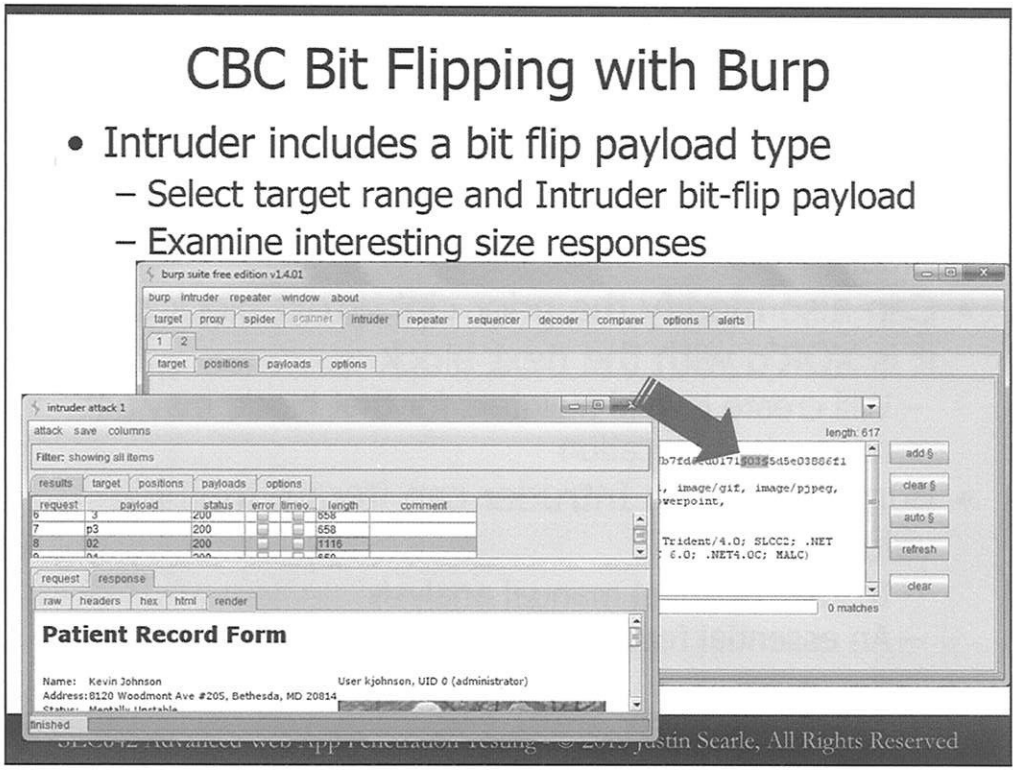
We should continue testing the system to determine what changes caused desirable effect on the system. Here, we want to change the UID to a value of "000" to escalate our system privileges.



Data Manipulation

Through experimentation we learned that the 10th byte of the IV influences the first byte of the decrypted UID value. With the user "kjohnson", we see that the UID is "100"; we want to change the UID to "000" to gain additional privileges on the system.

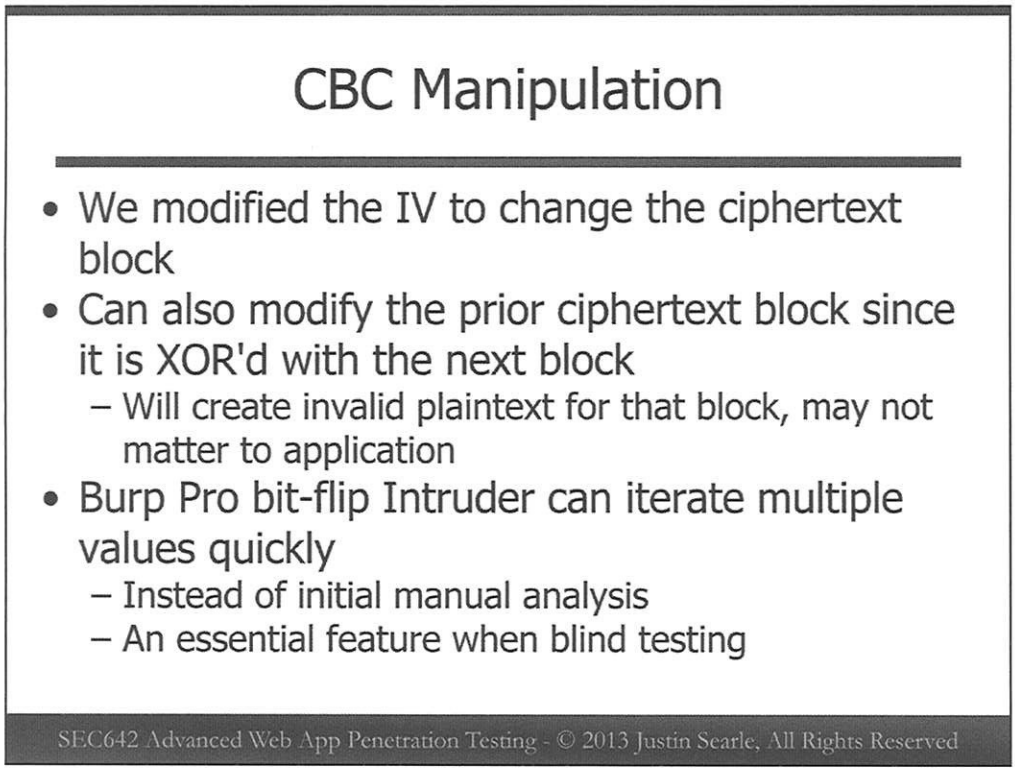
To change the UID to "000", we need to decrement the first plaintext byte of this field by one. Looking at the 10th byte of the IV we have a value of 0x03. Decrementing this value by one to 0x02 will cause the plaintext byte to be similarly decremented, producing a UID value of "000". The system interprets this string as a value of integer 0, further marking the account as an administrator. Now, if we attempt to change the `prec` parameter to access other patient information we will obtain a different patient record.



CBC Bit Flipping with Burp

We can perform CBC bit flipping manually (conveniently in the previous example by editing the URL, or using a proxy replay tool such as Burp Repeater for more complex data encoding methods), but we can also automate this process. Burp's Intruder includes a payload type of "bit flipper" which will flip the bits of marked data.

In the background image on this slide we marked the 10th byte of the IV for bit-flipping with Burp Intruder. We also changed the `prec` value to a different record to easily identify any responses that grant us access to information other than the insufficient privileges error message. In the Intruder attack result page (foreground) we can easily examine the length of the responses to identify the bit-flipped request that granted us additional access to the system.

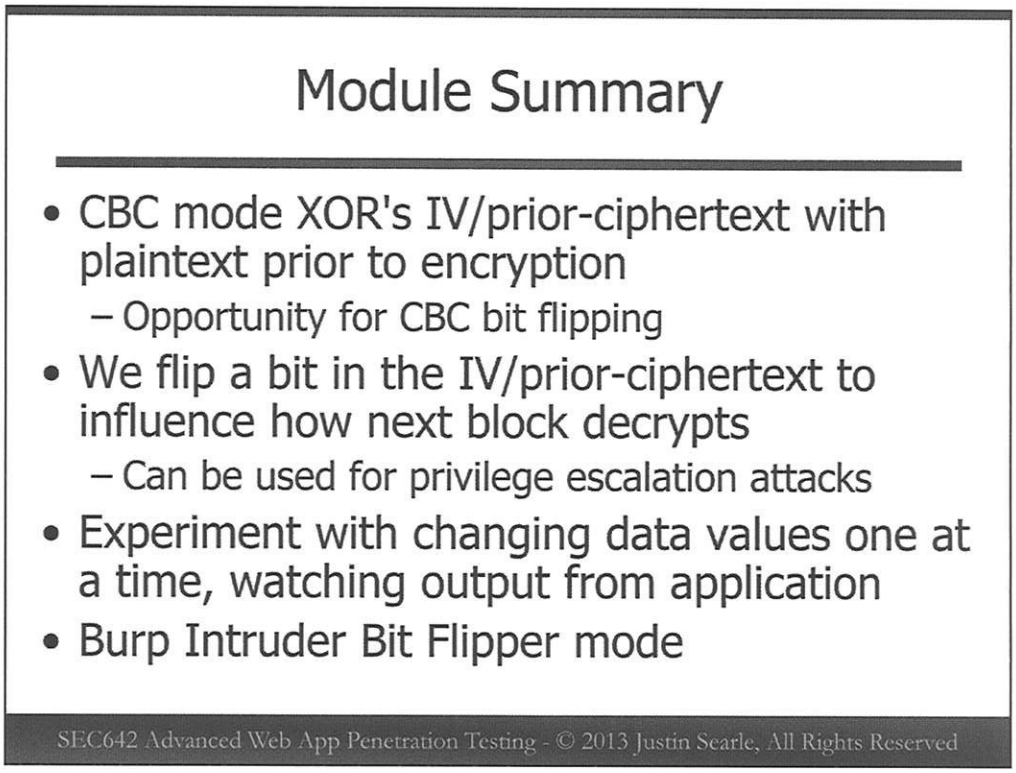


CBC Manipulation

In this example, we used CBC manipulation to gain additional access to the system by modifying the IV in order to change the first (and only) ciphertext block. This was very convenient, as changes to the IV do not negatively affect any of the encrypted blocks of data other than our target block.

If the content we wanted to manipulate was not in the first block, we would need to continue to modify the ciphertext blocks just as we did for the IV. When manipulating ciphertext blocks however, the decrypted response would be invalid for the byte we are modifying. Some applications will reject this immediately, preventing the attack from success unless we can make the ciphertext block modifications decrypt to meaningful data. Other applications may survive one bad block of decrypted data, depending on the level of error checking and exception handling present in the system.

In the demo application, we had the opportunity to see where our bit-flipped values were changing plaintext; this is not always possible, making it more difficult for us to recognize where our IV or ciphertext block changes create desirable application modifications. In this blind manner of CBC bit flipping, an automated tool that can quickly iterate through multiple blocks is very useful. While the free version of Burp can perform bit flipping for a large collection of blocks, it does so relatively slowly due to limitations in the tool. However, the Burp Pro version does not have this same set of limitations, allowing us to quickly test large blocks of encrypted content.

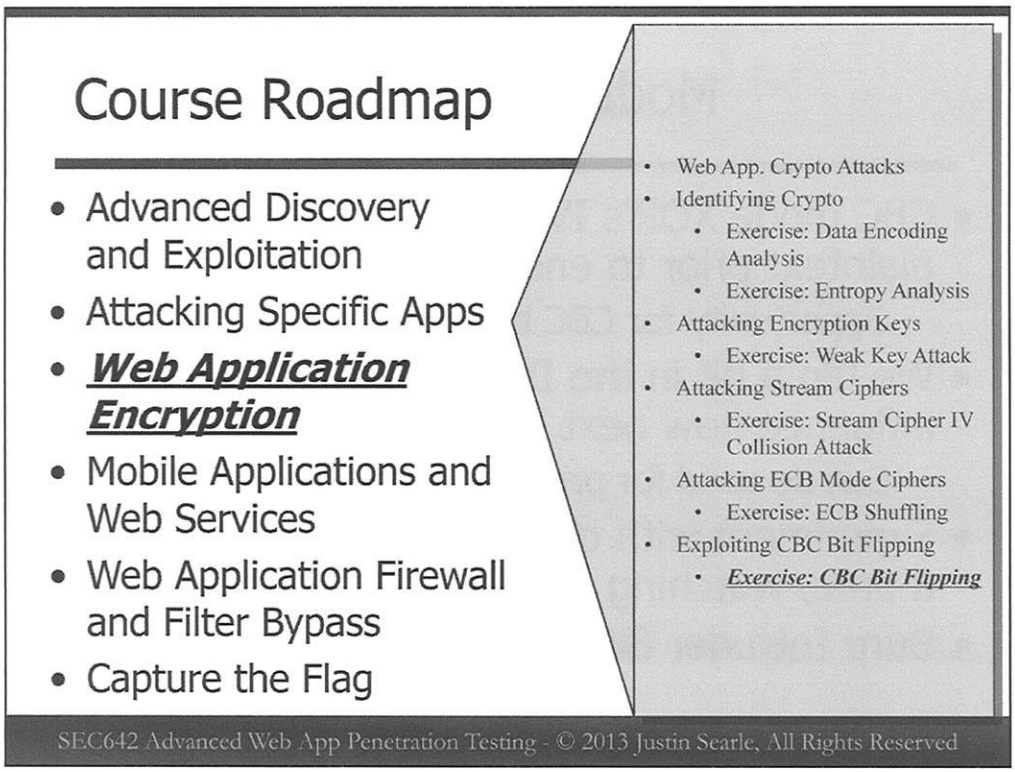


Module Summary

Recall that, in CBC mode, the prior block of ciphertext (or the IV for the first block) is XOR'd with the plaintext prior to encryption. This operation is what leads to the CBC bit flip attack opportunity, allowing us to manipulate decrypted values by changing the prior encrypted block (or the IV).

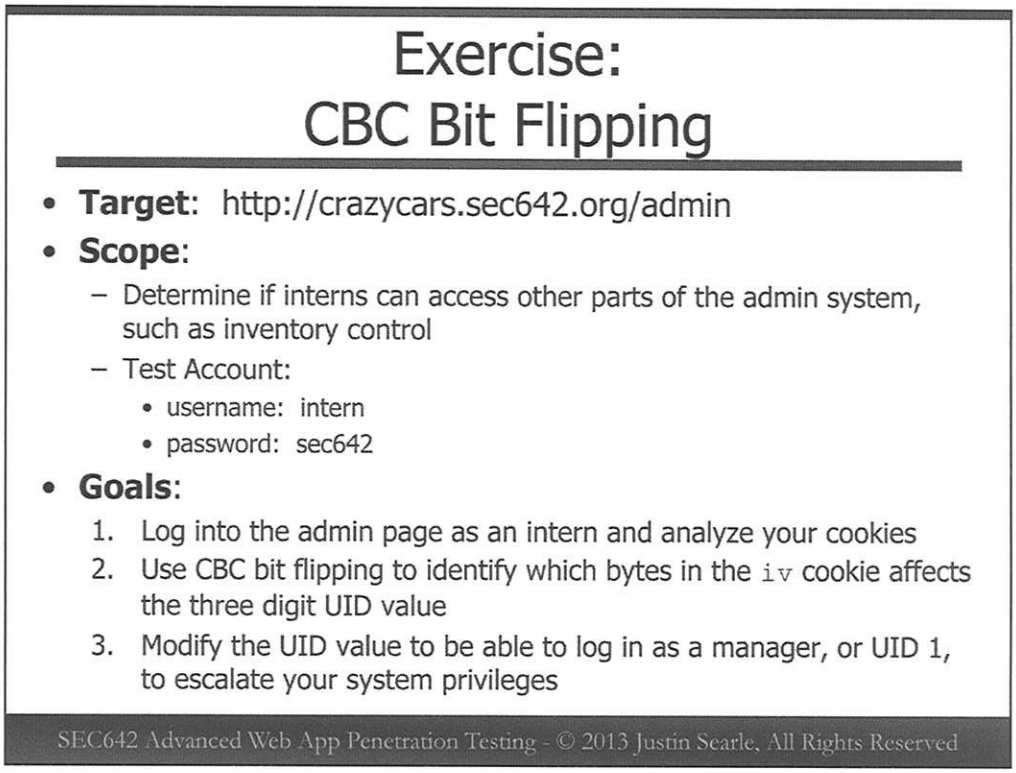
In testing CBC encrypted data, we can manually experiment with changing one byte of IV or encrypted block at a time and observing the application's response. This can lead to privilege escalation attacks, or other data manipulation opportunities in web applications.

Finally, Burp Intruder's bit flipper mode helps us automate this attack, quickly iterating through large blocks of encrypted data.



Course Roadmap

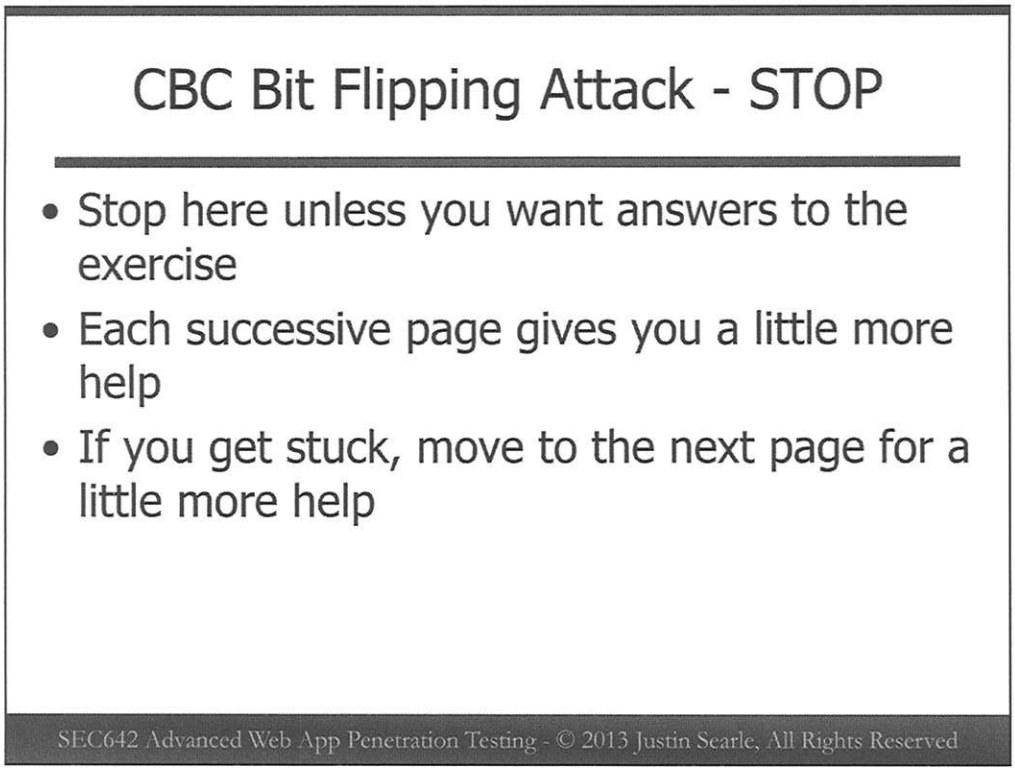
Now we will do an exercise.



Exercise: CBC Bit Flipping

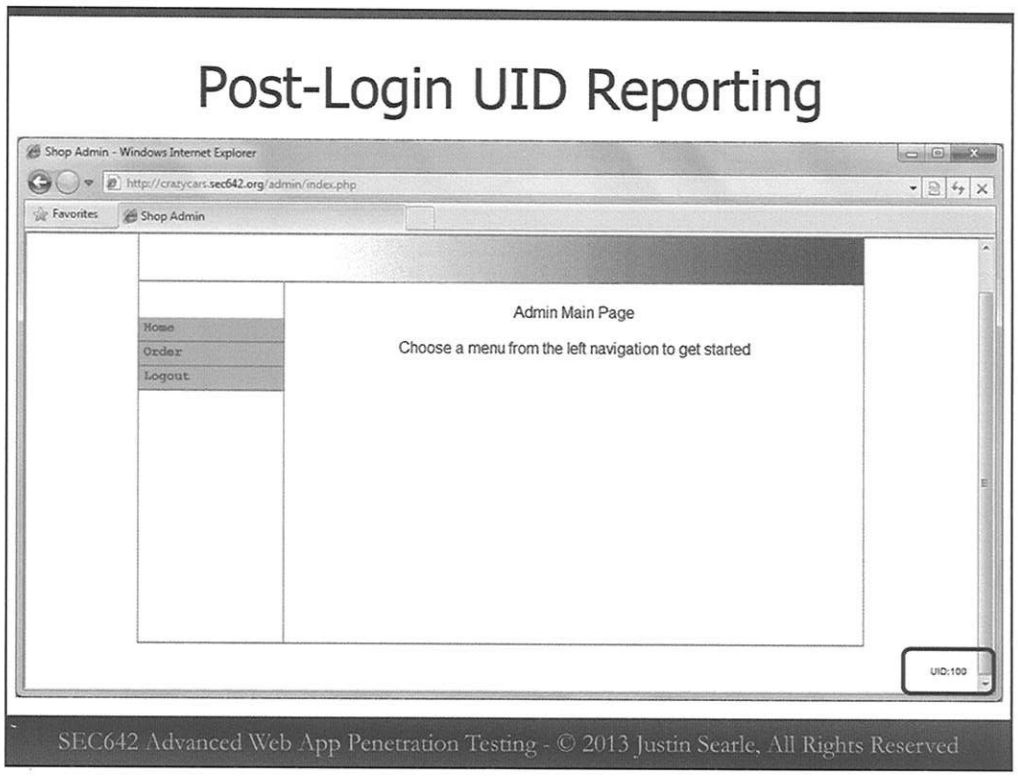
In this exercise you'll target the administrative login page for Josh's Crazy Antique Car Store at http://crazycars.sec642.org/admin. You are to test to see if interns at Josh's Crazy Cars can access other parts of the admin system, such as inventory control. Your login credentials are "intern" and "sec642" for a password.

You will use CBC bit flipping to escalate your system privileges. You will need a proxy tool to complete this exercise; you can use Burp or ZAP available at http://files.sec642.org.



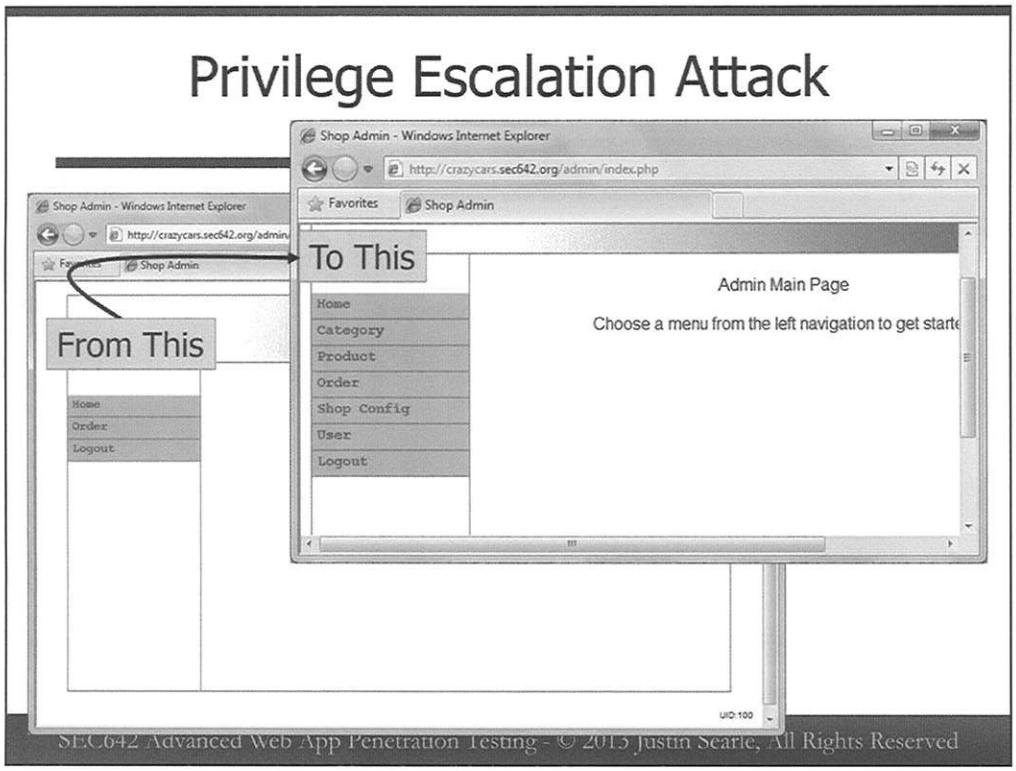
CBC Bit Flipping Attack – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



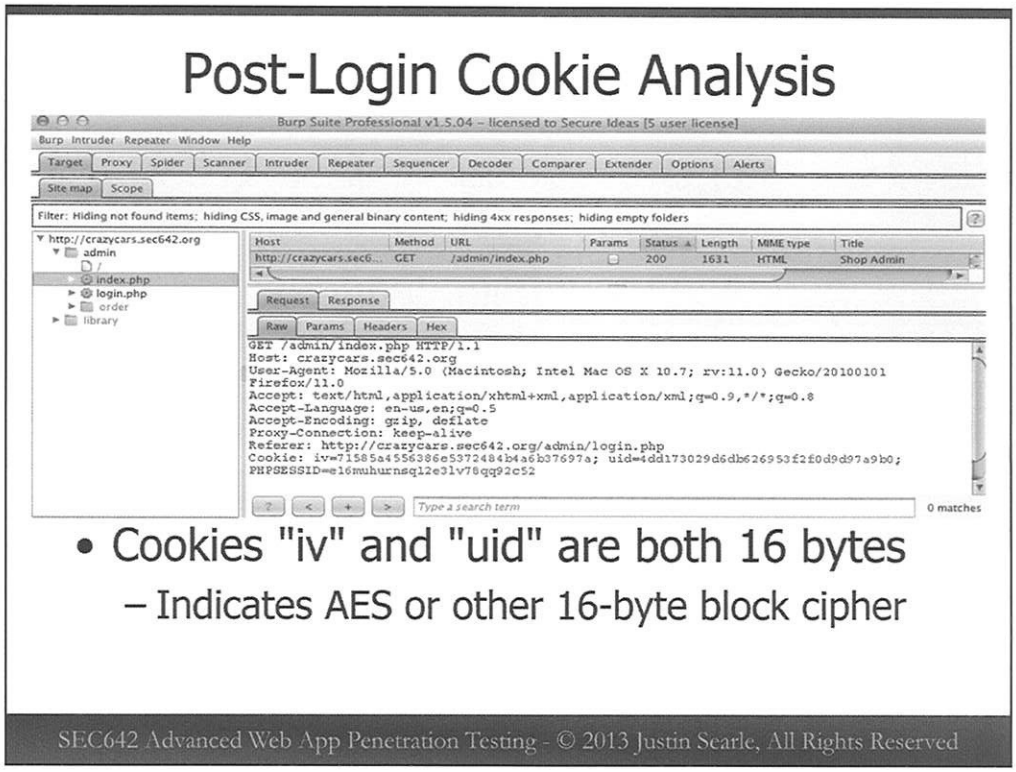
Post-Login UID Reporting

After logging in to the application, notice that the UID is displayed in the bottom-right corner of the application. You can observe this content as you perform the CBC bit flipping attack.



Privilege Escalation Attack

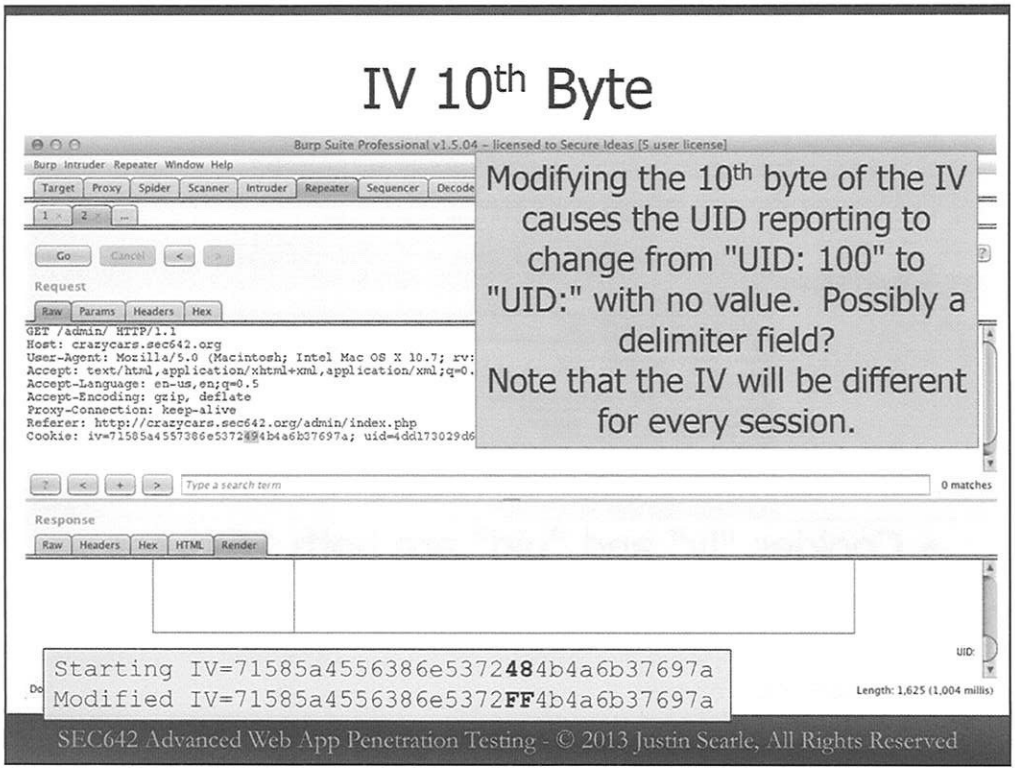
This slide shows your basic system access as an intern, with the "Home", "Order" and "Logout" tabs accessible. You want to escalate your system privileges to gain access to additional functions such as "Category", "Product", "Shop Config" and "User", as shown.



Post-Login Cookie Analysis

While viewing the transaction with Burp, after logging in we see several cookies set including "iv" and "uid", our attack targets. Both are 16-bytes likely indicating AES encryption or another 16-byte block cipher.

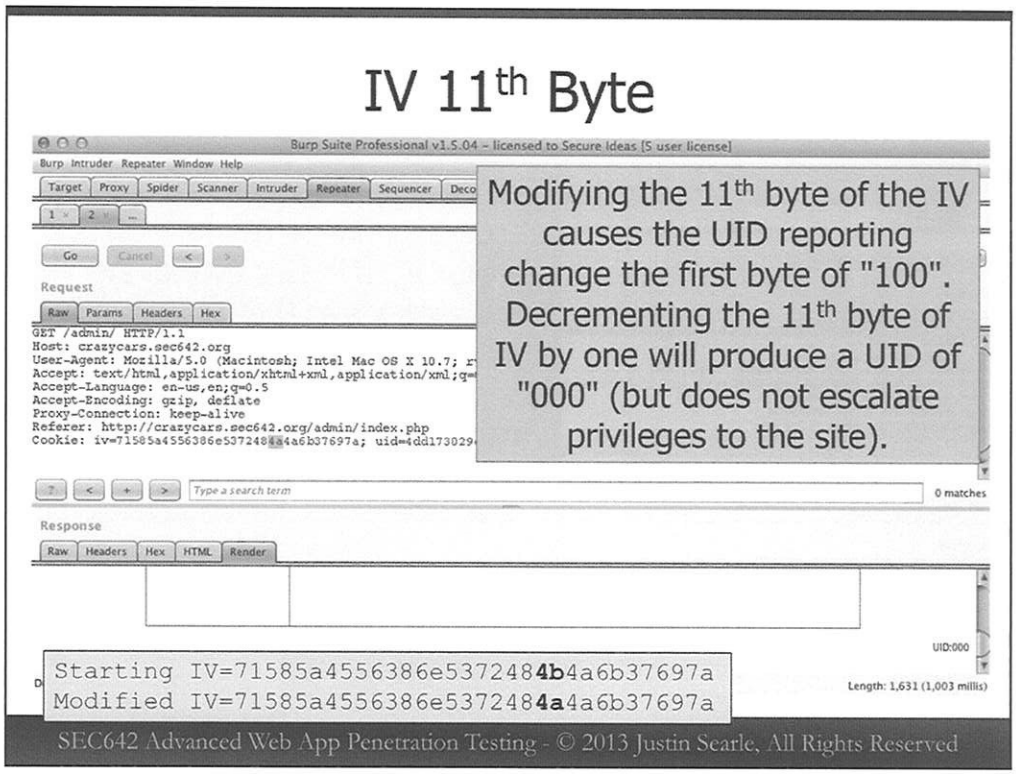
Send a subsequent request that includes the cookie content to repeater and start to manipulate the IV value one byte at a time until the UID content starts to look interesting.



IV 10th Byte

Modifying the 10th byte of the IV causes the UID reporting to change. Note that your IV will be different than what is shown here since it is randomly selected at the beginning of the session.

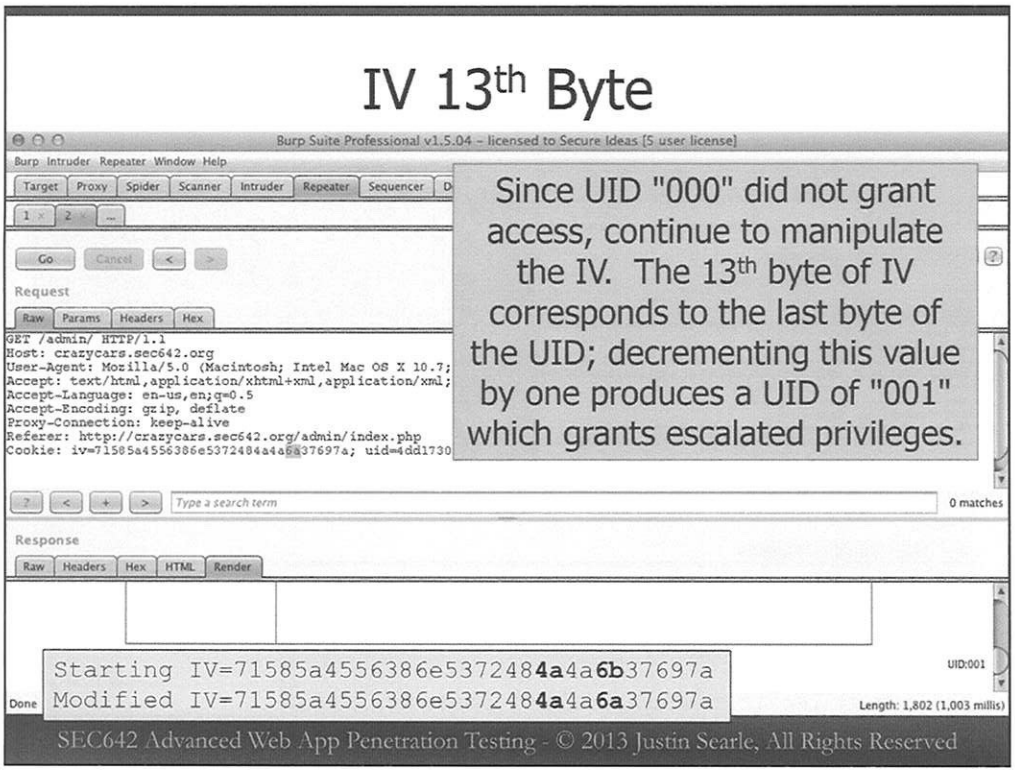
In the example here, changing the 10th byte of IV from 0x43 to 0xFF causes the value following the "UID:" label to change. Depending on your IV, this may cause the numeric identifier to disappear. This is likely the presence of a string delimiter that was overwritten, causing a server-side execution function such as a string split to fail.



IV 11th Byte

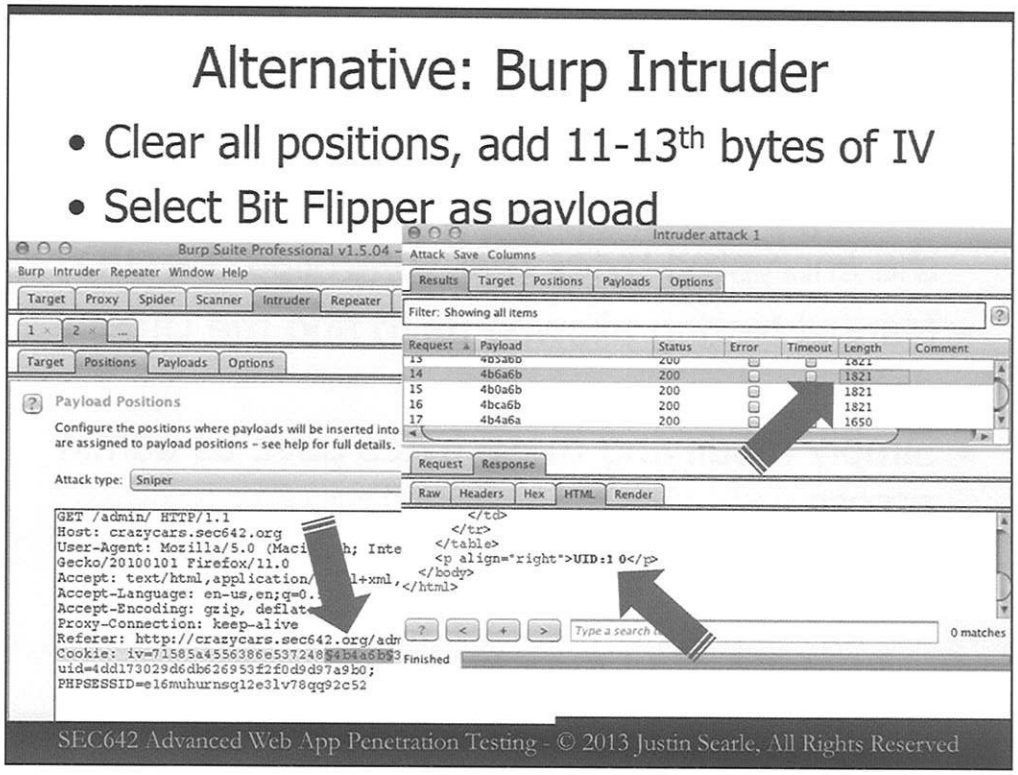
The 11th byte of the IV starts to manipulate the first byte of the numeric UID value of "100". If you decrement the IV value by one, the IV will now turn into "000".

Unfortunately, having a UID of "000" does not grant additional access to the site. We need to keep experimenting with the site to find a UID value that does grant escalated access.



IV 13th Byte

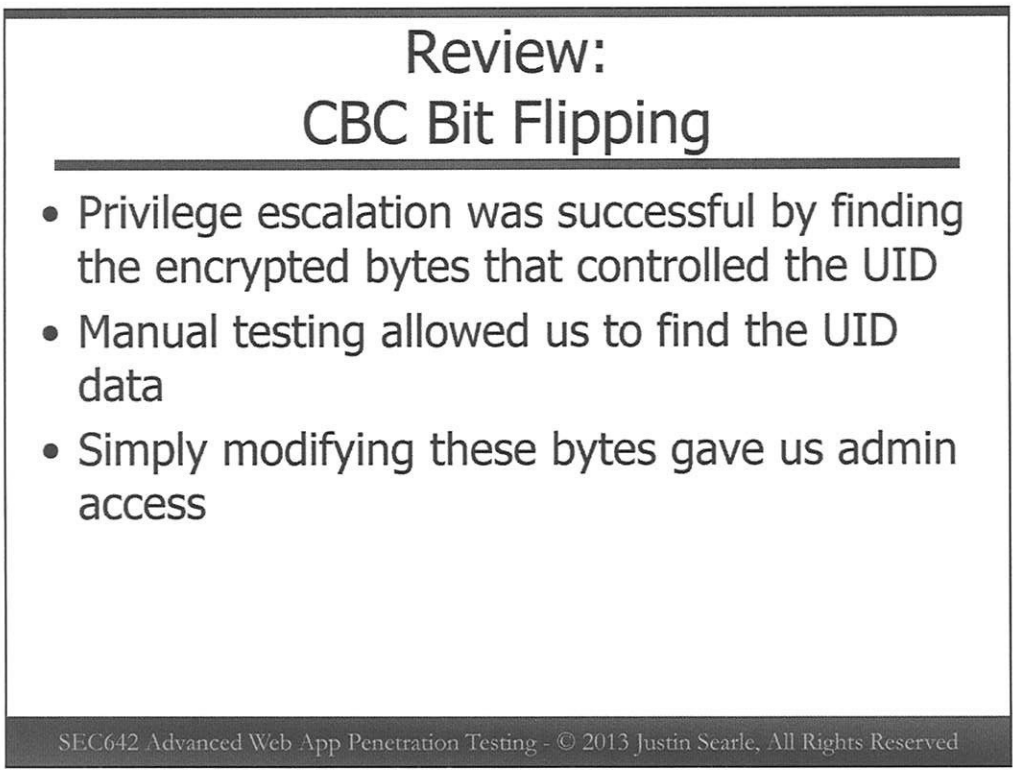
If you continue to experiment with the IV you will find that the 13th byte of the IV controls the last byte of the UID. Decrementing this byte of the IV (for this example) will create a UID of "001", granting escalated privileges to the site.



Alternative: Burp Intruder

As an alternative solution, instead of manually manipulating the byte values, we can use Burp Intruder's bit flipper payload function. Send the request to Intruder and clear all the payload positions. Next, add a payload position, focusing on the 11th through 13rd bytes of the IV.

Select a payload type of "bit flipper", then start the attack by clicking intruder | start. Look for a response length that is larger then the others to reveal additional access to the site.



This lab took some trial and error as we worked our way through the encrypted blocks in the *iv* cookie. This process was made easier by using Burp Repeater to modify the encrypted blocks and send the request over and over while watching how the application behaved. Once we were able to discover the location of the UID data in the ciphertext, we were able to use Burp Intruder to attack just that section of the cookie and elevate our privileges to admin.

Conclusions

- Today we have covered an overview of the use of cryptography in modern web applications
  - Identifying cryptography
  - Data encoding of cryptographic keys
  - Entropy analysis
  - Weak key attacks
  - Stream cipher attacks
  - ECB shuffling
  - CBC Bit Flipping
- Tomorrow we will finish our discussion of cryptographic weaknesses in web applications
  - CBC chosen plaintext
  - Padding oracle attacks
  - Crypto testing recommendations

Tomorrow, we will move continue our discussion of cryptography.