Common Pitfalls in Cryptography for Software Developers

Shay Zalalichin,
CISSP
AppSec Division Manager,
Comsec Consulting

shayz@comsecglobal.com

OWASP AppSec Israel

July 2006

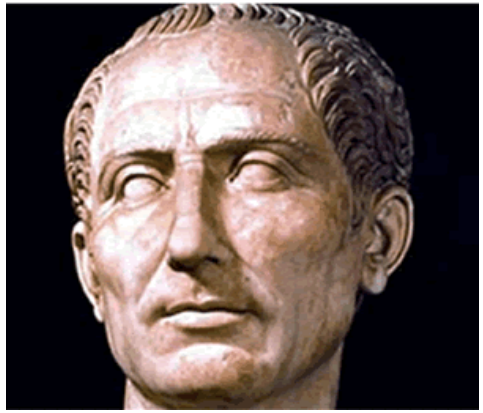
COMSEC Consulting
Information Security

The OWASP Foundation
http://www.owasp.org/

Agenda

- Demonstrating Crypto Failures
- Common Crypto Terminology
- The "Top 10" Crypto Pitfalls
- Conclusions

Example #1 – The Cesar Cipher



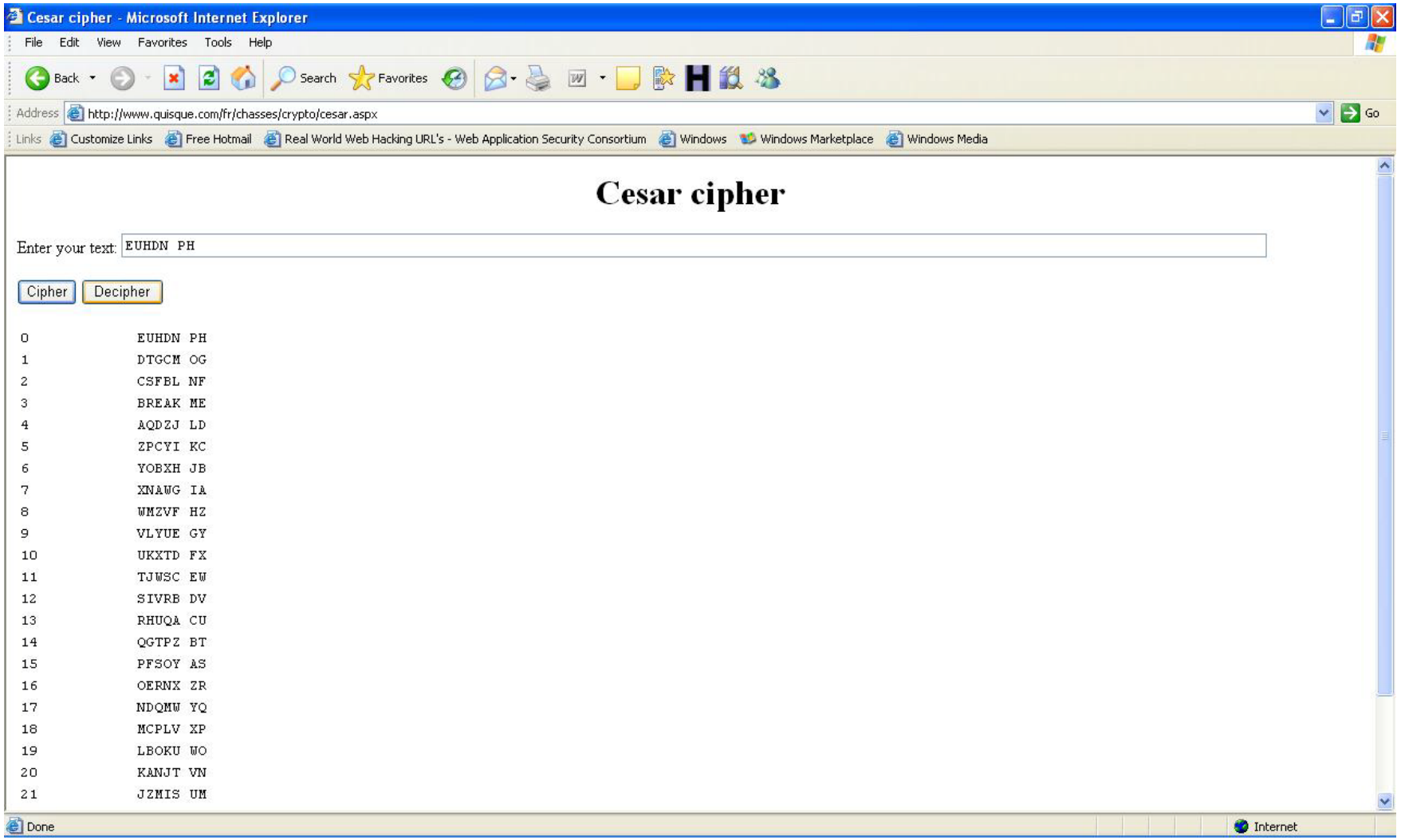
- Earliest known Substitution Cipher
- Invented by Julius Caesar, 49 BC
- Used to communicate with his army officers and keep the messages secure
- Replaces each letter by 3rd letter on

- example:
  - MEET ME AFTER OWASP LECTURE
  - PHHW PH DIWHU RZDVS OHFWXUH

Cryptanalysis of the Cesar Cipher

- Only have 26 possible ciphers
  - A maps to A,B,..Z
- Could simply try each in turn using a **Brute Force Search**
- Given Ciphertext, just try all shifts of letters
- Do need to recognize when have plaintext

Cryptanalysis of the Cesar Cipher



Example #2 – Using 'Complex' Substitution Cipher

Definition Let $\mathcal{A}$ be an alphabet of q symbols and $\mathcal{M}$ be the set of all strings of length t over $\mathcal{A}$. Let $\mathcal{K}$ be the set of all permutations on the set $\mathcal{A}$. Define for each $e \in \mathcal{K}$ an encryption transformation E_e as:

$$E_e(m) = (e(m_1)e(m_2)\cdots e(m_t)) = (c_1c_2\cdots c_t) = c,$$

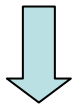
where $m = (m_1m_2\cdots m_t) \in \mathcal{M}$. In other words, for each symbol in a t-tuple, replace (substitute) it by another symbol from $\mathcal{A}$ according to some fixed permutation e. To decrypt $c = (c_1c_2\cdots c_t)$ compute the inverse permutation $d = e^{-1}$ and

$$D_d(c) = (d(c_1)d(c_2)\cdots d(c_t)) = (m_1m_2\cdots m_t) = m.$$

E_e is called a *simple substitution cipher* or a *mono-alphabetic substitution cipher*.

Example:

- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

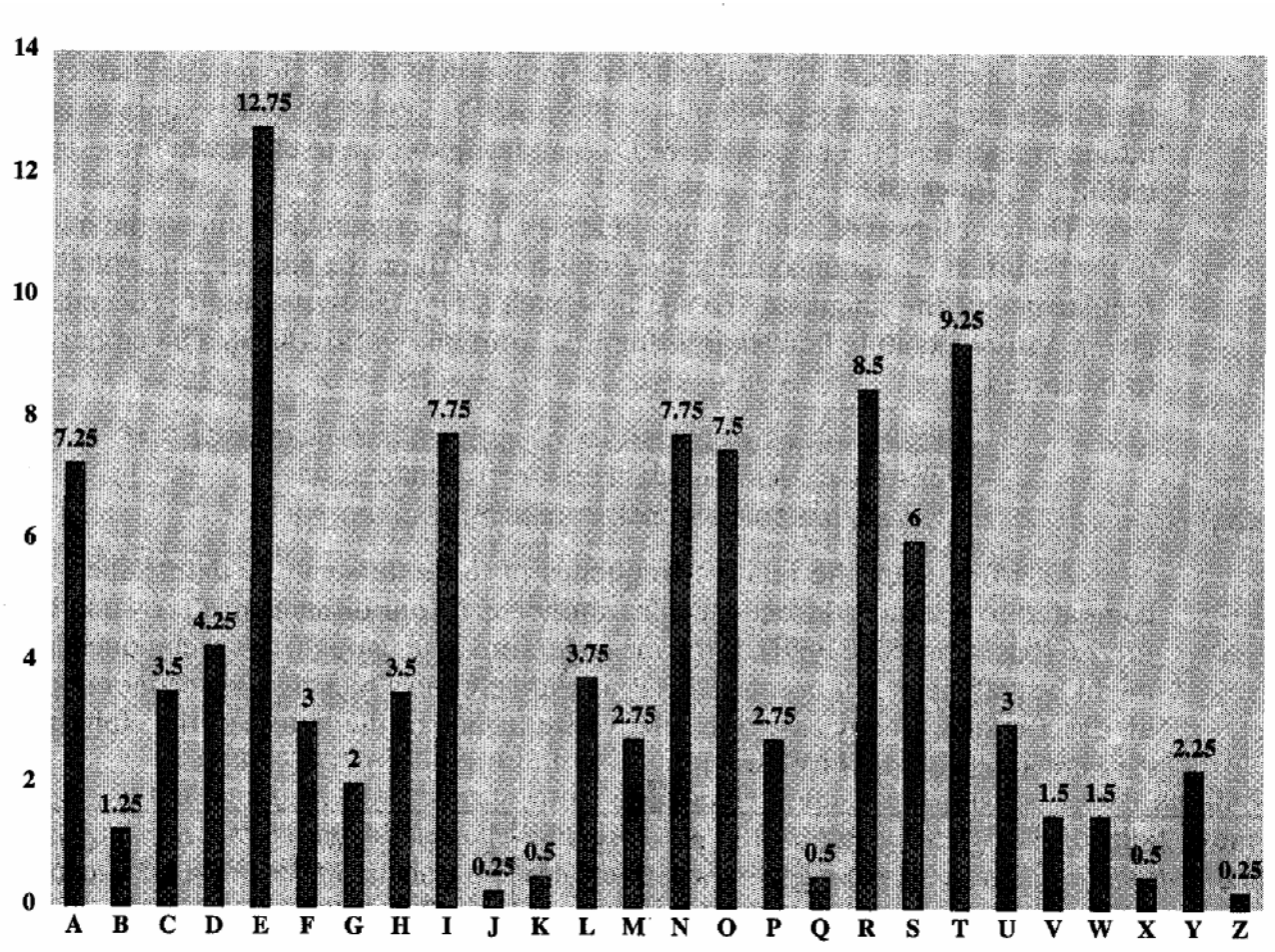


- D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Cryptanalysis of 'Complex' Substitution Cipher

- The key space for the English Alphabet is very large: $26! \approx 4 \times 10^{26}$
- It can be, however, broken easily, especially if the message space has known structure.
- For example, in English texts the letter "E" is the most used letter.
- Hence, if one performs a frequency count on the ciphers, then the most frequent letter can be assumed to be "E"
- Other letters are fairly rare (e.g. Z,J,K,Q,X)
- Have also tables of single, double & triple letter frequencies

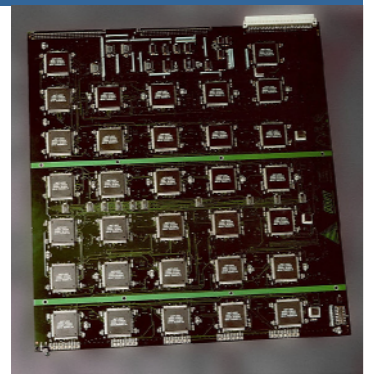
Cryptanalysis of 'Complex' Substitution Cipher



Cryptanalysis of 'Complex' Substitution Cipher

- Demo.

Example #3 – Data Encryption Standard



- US encryption standard [NIST 1993]
- 56-bit symmetric key, 64 bit plaintext input
- How secure is DES?
  - In 1977, Diffie and Hellman proposed a machine costing an estimated US$20 million which could find a DES key in a single day
  - By 1993, Wiener had proposed a key-search machine costing US$1 million which would find a key within 7 hours
  - In 1998 the EFF built Deep Crack which costs $250,000 and decrypted DES in 56 hours of work
  - In 1999 (Six months later) in collaboration with Distributed.net, Deep Crack decrypted DES in 22 hours and 15 minutes – **less then a day!**

Example #3 – Enhancing DES

- What if we want to 'enhance' its strength beyond of 2^56?
- We might consider 'enhancing' the algorithm to perform Double Encryption with two different 56-bit Keys
  - C = Ek1(Ek2(M))
- Total strength is considered to be: 2^112 (twice as strong as the original DES)

Example #3 – Breaking the Enhancement

- However …
- Time & Memory tradeoff attack called 'Meet-in-the-Middle' (from Diffie & Hellman) can be performed against the 'improved' algorithm breaking it in only 2^57 guesses
- Attack includes:
  - Using know P, C
  - Computing Ek1(P) for all k1
  - Computing Dk2(C) for all k2
    - Checking if Ek1(P) = Dk2(C) and if so, the key was found!

Example #4 – Using Public Key Mechanism

- Alice generates Ea & Da
- Alice sends Bob Ea
- Bob generates Eb & Db
- Bob sends Alice Eb
- Alice sends Bob encrypted messages using Eb and decrypts incoming messages using Da
- Bob sends Alice encrypted messages using Ea and decrypts incoming messages using Db

What can go wrong?

Common Pitfalls – Introduction

- Covers the popular mistakes & pitfalls that software developers do while using crypto within their applications
- Based on a few hundred security audits that Comsec performed over the last 6 years
- The list is not bound to any technology / application infrastructure or programming language

- Note: This list is far from being complete but serves its purpose to describe common mistakes

The "Top 10" List of Crypto Pitfalls

1. Security by Obscurity
2. Using Traditional Cryptography
3. Using Proprietary Cryptography
4. Using Insecure Random Generators
5. 'Hiding' Secrets
6. Using Weak Keys
7. Memory Protection
8. Not Using 'Salted' Hash
9. Using MAC Insecurely
10. Insecure Initialization

#1: Security by Obscurity

- One of the basic security principles that every software developer / designer should follow when writing secure applications
- In the crypto context, usually related to:
  - Hiding the Crypto algorithms / protocols being used
  - Hiding the Crypto keys generation algorithms
  - Hiding the Crypto keys
  - Hiding initialization parameters
  - Etc.
- Simply does not work … Once revealed, usually results in total breakdown of the system security model
- Side effect is the prevention of "other-eyes" from inspecting / reviewing the algorithms being used for potential weaknesses that can be fixed

#2: Using Traditional Crypto

- "Traditional" or "Classical" Cryptography covers algorithms and techniques used to perform cryptographic operations in the "old-days"
- Popular examples include:
  - Shift Cipher (e.g. Caesar / C3)
  - Substitution Cipher
  - Affine Cipher
  - Vigenere Cipher
  - Etc.
- Effective Cryptanalysis techniques / tools are considered common knowledge and usually result in the total breakdown of these algorithms

#3: Using Proprietary Crypto

- Includes the modification of standard crypto algorithms or the usage of proprietary new crypto algorithms
- Usually combined with #1 (Security by Obscurity)
- In most cases standard algorithms / methods cover most aspects needed by software developers

- Unless you have vast experience and significant resources to invest in the cryptanalysis research of the new algorithm – usually results in poor algorithm that can be easily broken

#4: Using Insecure Random Generators

- Usually Random Generators are not truly random but rather Pseudo Random algorithms
- Common usage of the random generators in the crypto context usually includes a generation of:
  - One-time-passwords
  - Access codes
  - Crypto keys
  - Session identifiers
  - Etc.

#4: Using Insecure Random Generators (Cont.)

- Some of the generators were planned to support statistical requirements (e.g. uniformly distribution) and not to provide secure unpredictable values
- Some of the generators are required to be initialized by using SEED value. Usage of the same SEED provides the same sequence – causing the choosing of the SEED to be critical for security

- Using random generators that are considered insecure or the insecure initialization of these generators usually results in the ability of an attacker in predict other values that were generated by the random generator

#5: 'Hiding' Secrets

- Secure systems need to handle secrets
- These secrets are usually access credentials to other systems (e.g. DBMS) or crypto keys (sometimes even to protect those access credentials)
- In many cases, to protect those secrets developers 'hides' them in several places such as:
  - Application source code
  - Configuration files
  - Windows Registry

#5: 'Hiding' Secrets (Cont.)

- It is common belief that since application code is 'compiled' into binary executable it would be hard to extract them
- The fact is that extracting those secrets from application 'compiled' code is eventually not as hard as it seems and can be done using various tools and techniques
- Some programming languages (e.g. Java, .NET) do not produce 'true' binary making the extraction job even easier

#5: 'Hiding' Secrets (Cont.)

- Additional problems with 'hard-coded' secrets are that they are hard to maintain and expose these secrets to developers or to less secure environments (e.g. Testing, Integration, Source Control)
- As for hiding the secrets in external resources (e.g. files), plenty of available tools can be easily used to discover the resource used for storage and easily recover the secret

#6: Using Weak Keys

- Systems that use crypto must use crypto keys to perform various crypto activities (e.g. for encryption, decryption, MAC)
- These crypto keys are sometime based on chosen passwords that might seem long & hard but when used as keys are weak
- Other cases include key generation using an insecure generator (e.g. pseudo random generator or other 'obscured' techniques)

#6: Using Weak Keys (Cont.)

- The strength of the entire crypto scheme is heavily dependent upon keys' strength
- It is important to notice that strength is not always equal to length
  - Example:
    - 8 char alpha numeric passwords strength is not equal to 64-bit crypto key but rather to 40-bit crypto key!
- Secure crypto schemes (e.g. AES, RSA) are almost useless and can be easily defeated when used with weak keys

#7: Memory Protection

- In most cases sensitive crypto data is stored in the host's memory
- Most of the time this information is stored unprotected (without any Encryption)
- Many times this information is stored for a longer period than actually needed (enlarging the attack window)
- Many times the memory is not specifically overwritten to perform the actual removal of the sensitive information from memory

#7: Memory Protection (Cont.)

- In some programming languages the information is even stored (usually due to insufficient knowledge) within a mutable object preventing it from being specifically erased …

- Insufficient memory protection jeopardizes the sensitive information (usually crypto Keys) to be 'leaked' out to an unauthorized attacker causing direct risk to the entire cryptosystem

#8: Not Using 'Salted' Hash

- Crypto Hash Functions are sometimes used to store 'one-way' version of Passwords (or other credentials that needs to be authenticated)
- Popular implementation includes a simple replacement of the Password field with the 'Hashed' version and the performance of the checks against it

- Not using any 'Salt' on the stored Hash Values exposes this scheme to Dictionary Attack that can totally break this scheme!

#9: Using MAC Insecurely

- MAC is used to secure the integrity of the message and to authenticate the sender by using a Shared Secret
- Popular MAC implementations are based on un-keyed Hashes operated on the data combined with the Shared Secret
- The Data + Shared Secret 'combination' is usually the concatenation operation

#9: Using MAC Insecurely (Cont.)

- Due to its mathematical operation of the iterative Hash function, if not carefully built, data can be altered and a matching MAC can be calculated without knowing the Shared Secret!

#10: Insecure Initialization

- Standard crypto algorithms and mechanisms are based on several parameters, most of them are defined during initialization
  - Examples include:
    - Symmetric block encryption modes of operation (e.g. ECB)
    - Asymmetric encryption (e.g. Modulus in RSA)
    - RSA Padding mode
- These parameters define the way the algorithms work and therefore have a direct impact on the algorithm security level
- Failure to choose them carefully may 'weaken' the algorithm exposing it to different types of attacks (e.g. MITM, Existential Forgery)

#10: Insecure Initialization – Example

- Usage of Shared Modulus in RSA results in total breakdown of the entire cryptosystem
  - If Bob has (N, Eb) & (N, Db)
  - And Alice has (N, Ea) & (N, Da)
  - Bob can factor N and then calculate Da from Ea

Conclusions

And if you can only take a few thing from this presentation:

- Know what you are doing
- Do not rely on 'Obscurity'
- Do not try to 'Hide' secrets
- Do not re-invent the wheel
- Generate Strong Keys and Protect them
- Use only strong & standard Ciphersuites

Questions?

Shay Zalalichin,
CISSP
AppSec Division Manager,
Comsec Consulting

shayz@comsecglobal.com

OWASP AppSec Israel

July 2006

COMSEC Consulting
Information Security

The OWASP Foundation
http://www.owasp.org/