

an

ORACLE®

CON7402 – ‘Heartbleed’ (CVE-2014-0160) Case Study I

Secure Development Perspective

John Heimann – Vice President, Security Program Management
Eric Maurice – Director, Oracle Security Assurance



Program Agenda

- 1 What is the 'Heartbleed' SSL vulnerability?
- 2 How did it come about?
- 3 What are the lessons learned?
- 4 What is the implication for open source development?
- 5 What lessons can YOU derive for YOUR organization?

DISCUSSING

- Today's discussion came about as a unique opportunity
- We believe that any organizations can derive significant lessons from 'Heartbleed':
 - Secure development ~~practices~~ ^{CTI} practices
 - Security testing
 - Software acquisition and use of open source components
 - Security responses policies and procedures
- This presentation is the first of a 2-part session:
 - Heartbleed case study I: Secure Development Perspective
- **NOW**
 - Heartbleed case study II: Vulnerability Handling Perspective – **Thursday, 2:30PM**

Program Agenda

- 1 What is the *'Heartbleed'* SSL vulnerability?
- 2 How did it come about?
- 3 What are the lessons learned?
- 4 What is the implication for open source development?
- 5 What lessons can YOU derive for YOUR organization?

CVE-2014-0160 a.k.a. Heartbleed



What is it?

- A vulnerability affecting certain versions of the OpenSSL crypto libraries.
 - OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable
- A successful exploitation can result in allowing malicious attacker with the ability to remotely (over the Internet) read (sections of) the memory of the targeted system:
 - Possible compromise of secret keys and other sensitive information
- It was called 'Heartbleed' because the bug originated in the OpenSSL's implementation of the TLS/DTLS (transport layer security protocols) heartbeat extension

Lets look at some code.

- <http://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed/>

This vulnerability was a big deal...

- Very prevalent use of OpenSSL – the open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1)
- These are cryptographic protocols intended to secure communications over the Internet
 - Web browsing
 - Emails
 - VoIP
 - Etc.

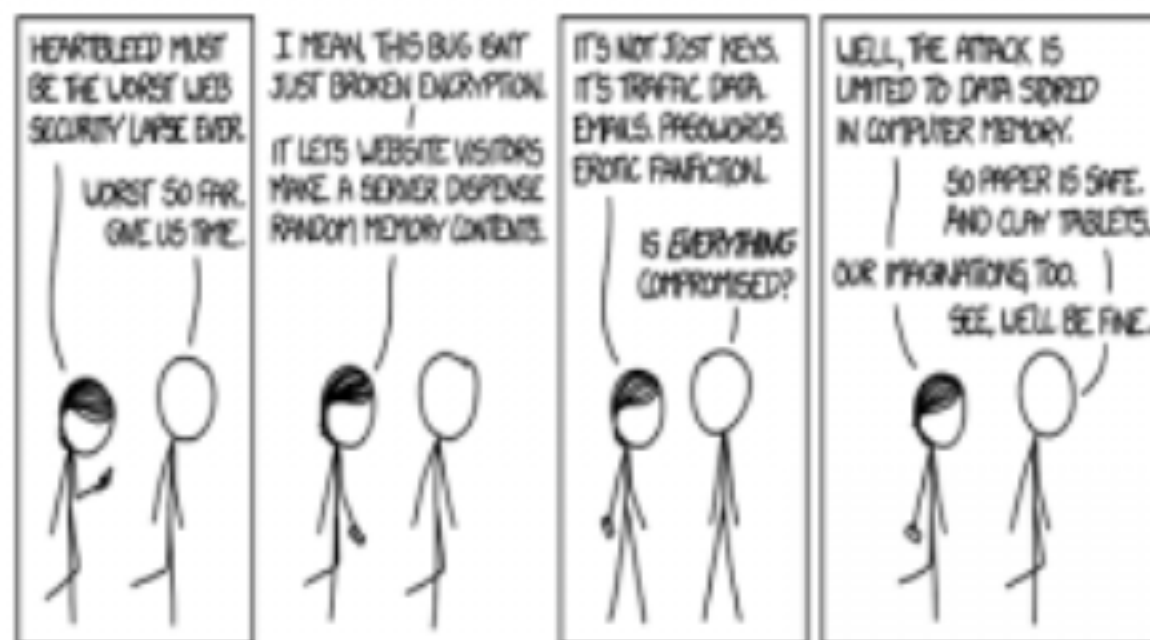


Source:

<http://xkcd.com/1354/>

... But the world didn't come to an end!

- Tremendous visibility in the press and security community resulted in quick response
- The vulnerability was limited to certain versions of OpenSSL
- The common use of SSL termination proxy (particularly with large sites) provided some level of mitigation against successful exploitation



Source:

<http://xkcd.com/1353/>

Program Agenda

- 1 What is the *'Heartbleed'* SSL vulnerability?
- 2 How did it come about?**
- 3 What are the lessons learned?
- 4 What is the implication for open source development?
- 5 What lessons can YOU derive for YOUR organization?



“In one of the new features, unfortunately, I missed validating a variable containing a length... In this case, it was a simple programming error in a new feature, which unfortunately occurred in a security relevant area”

– Robin Seggelmann

(Source: Sidney Morning Herald)

How was it discovered?

- Google and Codenomicon independently reported Heartbleed in April 2014
- OpenSSL may have been discovered and exploited by others who did not report it
 - OpenSSL code is open source and available for review by both well-intentioned and malicious analysts
 - A researcher review of audit logs suggested that hackers may have been aware of and may have exploited Heartbleed for at least five months prior to April 2014
- Hackers were definitely exploiting Heartbleed within days after it was reported

Program Agenda

- 1 What is the *'Heartbleed'* SSL vulnerability?
- 2 How did it come about?
- 3 What are the lessons learned?**
- 4 What is the implication for open source development?
- 5 What lessons can YOU derive for YOUR organization?

Thou shall not trust user input

- User input may
 - Have the wrong value
 - Have the wrong size or format
 - Be from an authorized but malicious user
 - Be from an unauthorized user (or have been modified by an malicious user)
- User input must be validated in form and content
 - Security functions should never rely on unvalidated user input
 - Malicious input to functions which aren't themselves security related can still cause security failures

Vulnerabilities due to unvalidated user input

- Buffer overflow
- Integer overflow
- SQL Injection
- Character encoding attacks
- Cross site scripting
- Cross site request forgery
- ...
- Heartbleed

2013 Table of Contents
2013 Top 10 List

A1 Injection	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2 Broken Authentication and Session Management	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.
A3 Cross-Site Scripting (XSS)	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A4 Insecure Direct Object References	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without proper access control checks or other protection, attackers can manipulate these references to access unauthorized data.
A5 Security Misconfiguration	Web security requires having a secure configuration defined and deployed for the application, framework, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6 Sensitive Data Exposure	Many web applications do not properly protect sensitive data, such as credit cards, log IDs, and authentication credentials. Attackers may steal or modify such easily protected data to conduct phishing and fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7 Missing Function Level Access Control	Most web applications verify function-level access rights before acting that functionality visible in the UI. However, applications tend to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.
A8 Cross-Site Request Forgery (CSRF)	A CSRF attack forces a legitimate victim's browser to send a forged HTTP request, including the victim's session cookies and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application treats as legitimate requests from the victim.
A9 Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or remote takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.
A10 Unvalidated Redirects and Forwards	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.



Program Agenda

1 What is the *'Heartbleed'* SSL vulnerability?

2 How did it come about?

3 What are the lessons learned?

4 What is the implication for open source development?

5 What lessons can YOU derive for YOUR organization?

Questioning closely-held assumptions

- 1,000,000 eyes, really?
 - Quality of code is directly related to the skills of the programmers
 - Code review doesn't happen automatically
 - Publication of code doesn't mean it is getting reviewed
 - What are the skills of the reviewers?
 - Security assurance requires time, effort and money (tools, experts, etc.)
- Open source vs. security by obscurity
 - Empowerment of hackers through public visibility into the code?
 - Unique challenges of coordinated disclosure with open source components
- It's really ALL ABOUT the level of trust you have in the producer(s) of the code

Open Source can be High Assurance

- Requires investment in security to make it secure
- ~~MySQL, Oracle Linux, Open Solaris, and OracleVM~~ are open source
 - Development and support teams are committed the security of the product
 - Follow the same OSSA security practices that all Oracle products are required to follow

Java Standard Edition (SE) is an interesting case study

- Prior to Oracle's acquisition of Sun, Java SE relied heavily on the "million eyes" of the community for security
 - Some vulnerabilities were found and fixed through the community process
 - Many more were found by professional researchers, and made public after Oracle's acquisition of Java
- Since Oracle's acquisition, Java has adopted Oracle security practices
 - Java development process has much greater pre-release security analysis and testing
 - Java security architecture has been improved
 - Better tools to manage Java security and remove old, vulnerable versions Java
 - *Significantly fewer vulnerabilities have been reported in Java*
- Java is still open source, but security improvements have come from Oracle's active investment in Java security



Program Agenda

- 1 What is the '*Heartbleed*' SSL vulnerability?
- 2 How did it come about?
- 3 What are the lessons learned?
- 4 What is the implication for open source development?
- 5 What lessons can YOU derive for YOUR organization?**

Key take-aways

1. Know where ~~your code~~ is coming from, and...
2. Assess the security assurance practices associated with anything used in your environment
 - COTS, open source, internally developed applications, etc.
3. Strong security assurance practices should include
 - Implementation of core security standards
 - ~~Security testing~~
 - Effective security vulnerability remediation procedures

ORACLE®

ORACLE®



Secure Software Development Securing Systems

Objectives

- Explain how SQL injection occurs for an SQL statement
- Explain OS Command Injection
- Explain why scrubbing of memory helps to secure a system.
- Draw a picture explaining how cross site scripting occurs.
- Explain a man in the middle attack
- Explain how comments may compromise security
- Explain how security misconfiguration can impact system security
- Define blacklist and whitelist
- Explain techniques that can be used to protect memory.
- List the secure code characteristics

Injection Flaws

```
String sSQLQuery = "SELECT * FROM USERS WHERE user_id = " +  
    txtUSERID.Text + " ` AND user_password = `" + txtPassword.Text  
    + "`"
```


OS Command Injection

- <http://www.mycompany.com/sensitive/cgi-bin/userData.pl?doc=%20%3B%20/bin/ls%20-l%20>

```
#include <iostream>

int main(int argc, char *argv[])
{
    char szTemp1[] = "After";
    char szTemp[16];
    char szTemp2[] = "Before";
    std::cin >> szTemp;
    std::cout << szTemp1 << "\n";
    std::cout << "Input string " << szTemp << "\n";
    std::cout << szTemp2 << "\n";
}
```

C++ STL >> A better way

```
#include <iostream>

int main(int argc, char *argv[])
{
    char szTemp1[] = "After";
    char szTemp[16];
    char szTemp2[] = "Before";
    std::cin.width(16);
    std::cin >> szTemp;
    std::cout << szTemp1 << "\n";
    std::cout << "Input string " << szTemp << "\n";
    std::cout << szTemp2 << "\n";
}
```

C++ STL >> Even better yet

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
    string szTemp1 = "After";
    string szTemp;
    string szTemp2 = "Before";

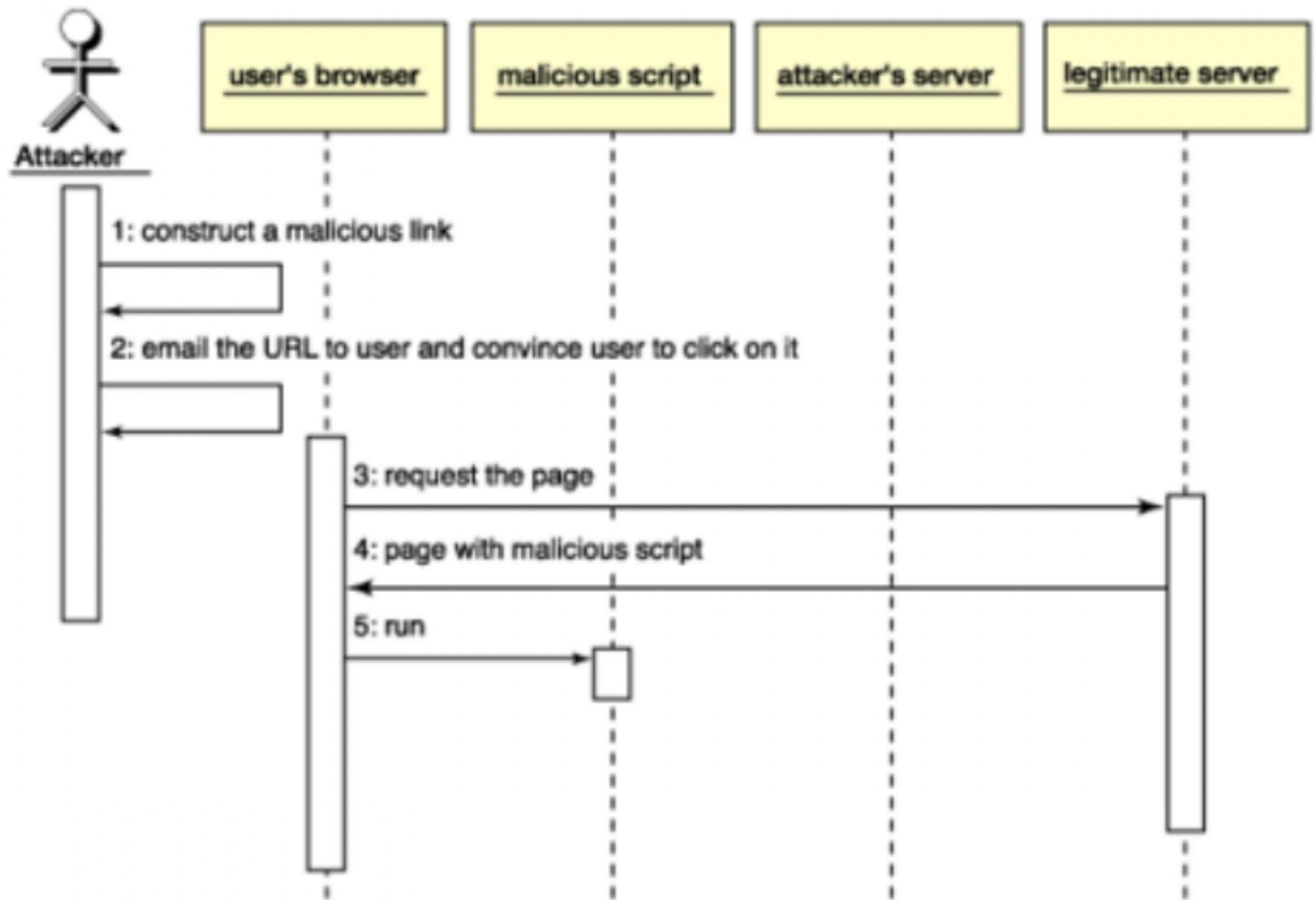
    cin >> szTemp;
    cout << szTemp1 << "\n";
    cout << "Input string " << szTemp << "\n";
    cout << szTemp2 << "\n";
}
```


Scrubbing Memory

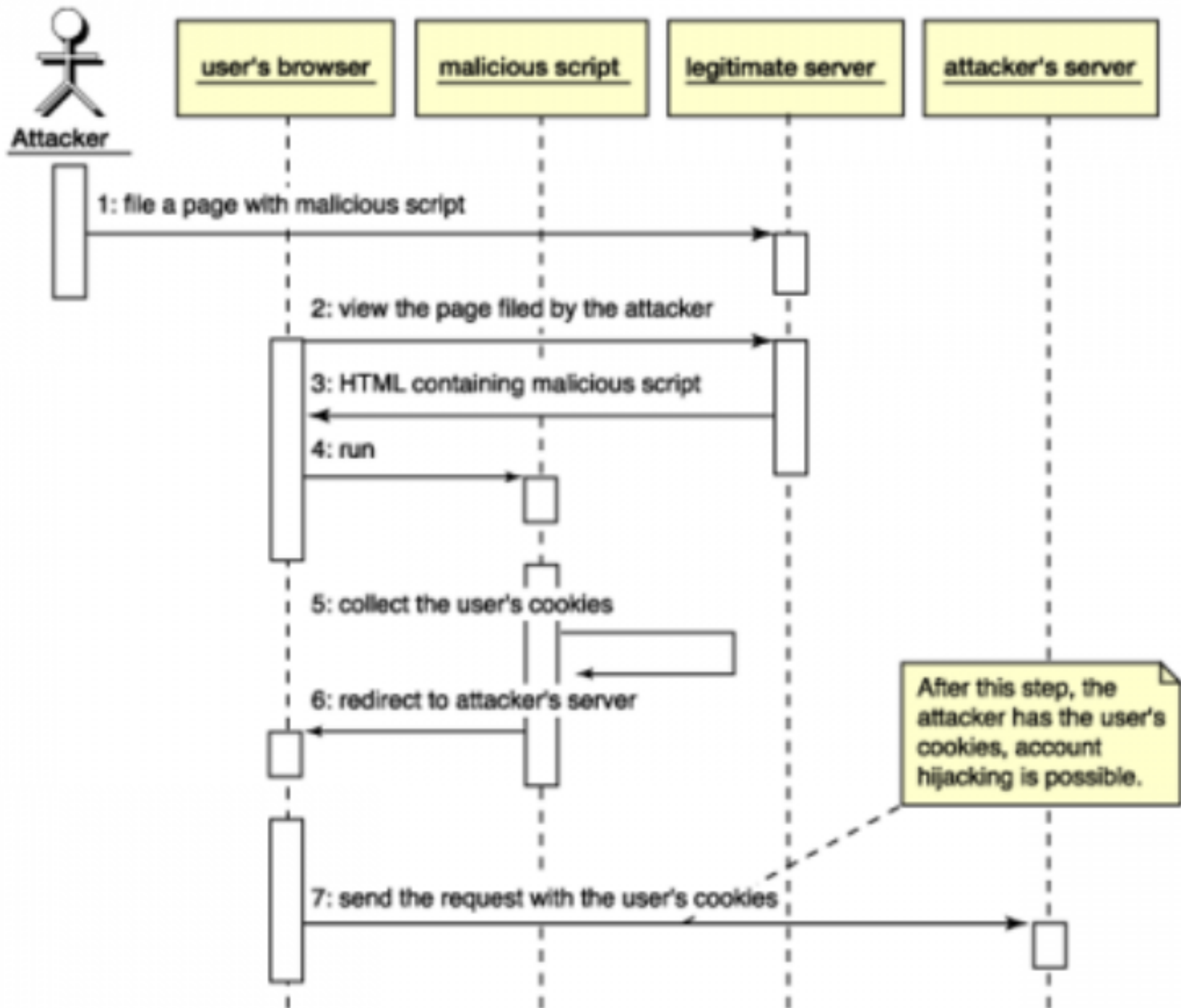
- [...]
memset(key, 0, 32);
[...]
- Fixed
 - [...]

```
void *secureMemset(void *v,int c,size_t n) {  
volatile char *p = v;  
while (n--)  
    *p++ = c;  
return v;  
}  
[...]  
secureMemset(key, 0, 32);  
[...]
```

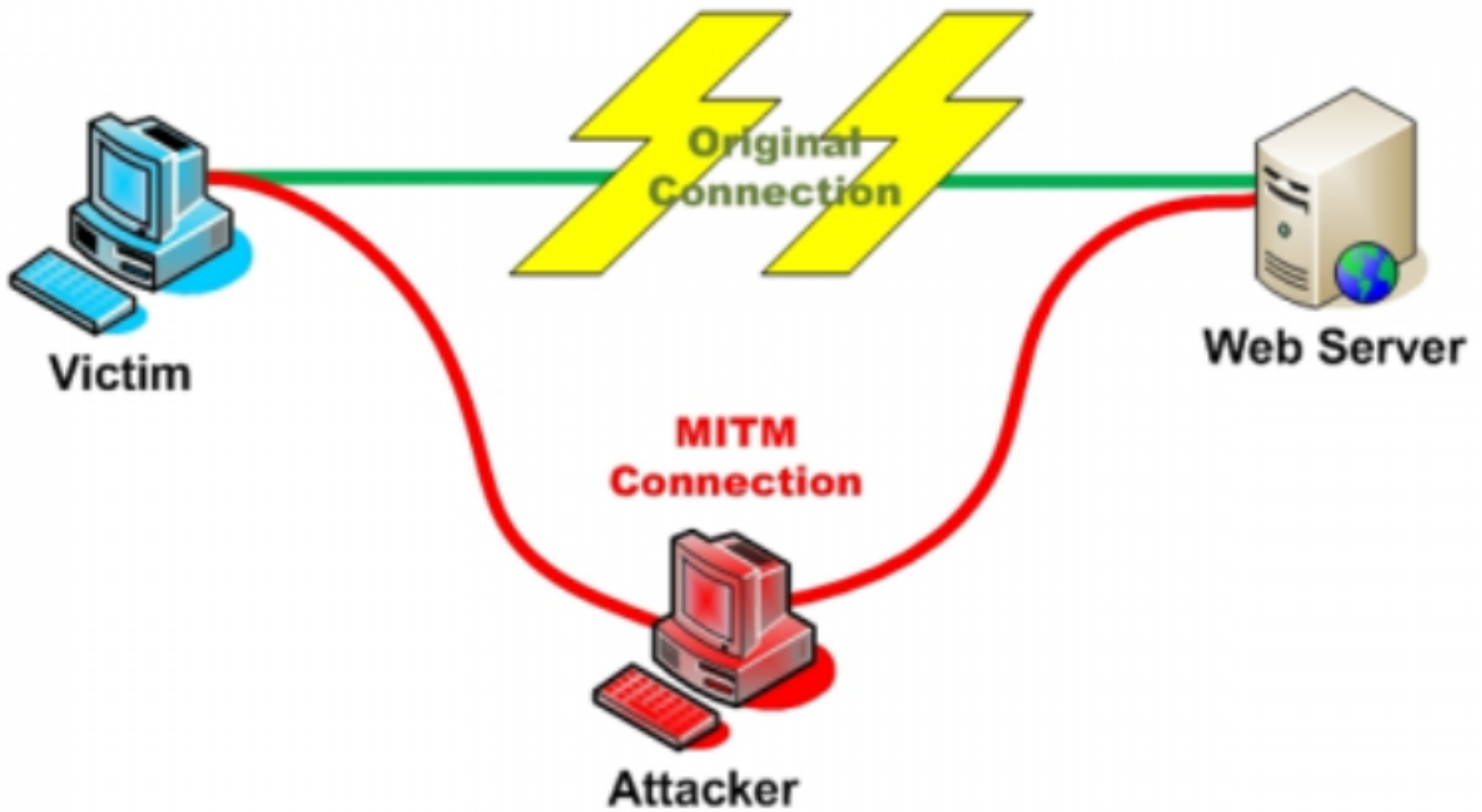
Cross Site Scripting



Cross Site Scripting



Man in the Middle Attack



Secure All Links



Security Related Comments in code...

```
/// <summary>
/// Establishes the connection to the database
/// </summary>
/// <param name="p_sServerName">Name of the database server.
/// Use HAMMERHEAD for Test and GREATWHITE for Production.
/// </param>
/// <param name="p_sLoginAccount">Name of the database login account.
/// Use Sally for Test and McQueen for Production.
/// </param>
/// <param name="p_sPassword">The password for the database login account.
/// Use Doc for Test and Mater for Production.
/// </param>
/// <param name="p_sDatabaseName">Name of the database to connect to.
/// Use PIKAR for Test and DREAMWORKS for Production.
/// </param>
private void BuildConnection(string p_sServerName,
    string p_sLoginAccount,
    string p_sPassword,
    string p_sDatabaseName)
{
    StringBuilder _oSBConnection = new StringBuilder();
    _oSBConnection.Append("Server=" + p_sServerName + ";");
    _oSBConnection.Append("uid=" + p_sLoginAccount + ";");
    _oSBConnection.Append("pwd=" + p_sPassword + ";");
    _oSBConnection.Append("Database=" + p_sDatabaseName + ";");

    SqlConnection _oSqlConn = new SqlConnection(_oSBConnection.ToString());
    _oSqlConn.Open();
}
```

Security Misconfiguration

- Hard Coded Credentials
- Not disabling the listing of directories or files on a web server
- Using default settings at installation
- Missing software patches
- Lack of perimeter controls

Defensive Coding Techniques

- Input validation
 - Regular Expressions
- Filtration
 - Whitelist
 - Valid and non-malicious characters allowed in a string
 - Blacklist
 - Characters which are not allowed in an input field

Memory Management

- Locality of Reference
 - Temporal, spatial, branch, or equidistant locality
- Dangling Pointers
 - Pointers to memory that has been freed
- Wild Pointers
 - Pointers not yet assigned
- Address Space Randomization
 - Memory allocator from operating systems
- Data Execution Prevention / Executable Space Protection
 - Harvard versus Von Neumann
- Stack Guarding

Secure code characteristics

Source code characteristics

- Validates input.
- Does not allow dynamic construction of queries using user-supplied data.
- Audits and logs business-critical functions.
- Is signed to verify the authenticity of its origin.
- Does not use predictable session identifiers.
- Does not hard-code secrets inline.
- Does not cache credentials.
- Is properly instrumented.
- Handles exceptions explicitly.
- Does not disclose too much information in its errors.
- Does not reinvent existing functionality, and uses proven cryptographic algorithms.
- Does not use weak cryptographic algorithms.
- Uses randomness in the derivation of cryptographic keys.
- Stores cryptographic keys securely.
- Does not use banned APIs and unsafe functions.
- Is obfuscated or shrouded.
- Is built to run with least privilege.