CT437
COMPUTER SECURITY AND FORENSIC COMPUTING

BUFFER OVERFLOW CASE STUDY – THE HEARTBLEED BUG

Dr. Michael Schukat

OÉ Gaillimh
NUI Galway

A Bug with its own Website (heartbleed.com) and Icon

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

What leaks in practice?

We have tested some of our own services from attacker's perspective. We attacked ourselves from outside, without leaving a trace. Without using any privileged information or credentials we were able steal from ourselves the secret keys used for our X.509 certificates, user names and passwords, instant messages, emails and business critical documents and communication.

How to stop the leak?

As long as the vulnerable version of OpenSSL is in use it can be abused. Fixed OpenSSL has been released and now it has to be deployed. Operating system vendors and distribution, appliance vendors, independent software vendors have to adopt the fix and notify their users. Service providers and users have to install the fix as it becomes available for the operating systems, networked appliances and software they use.

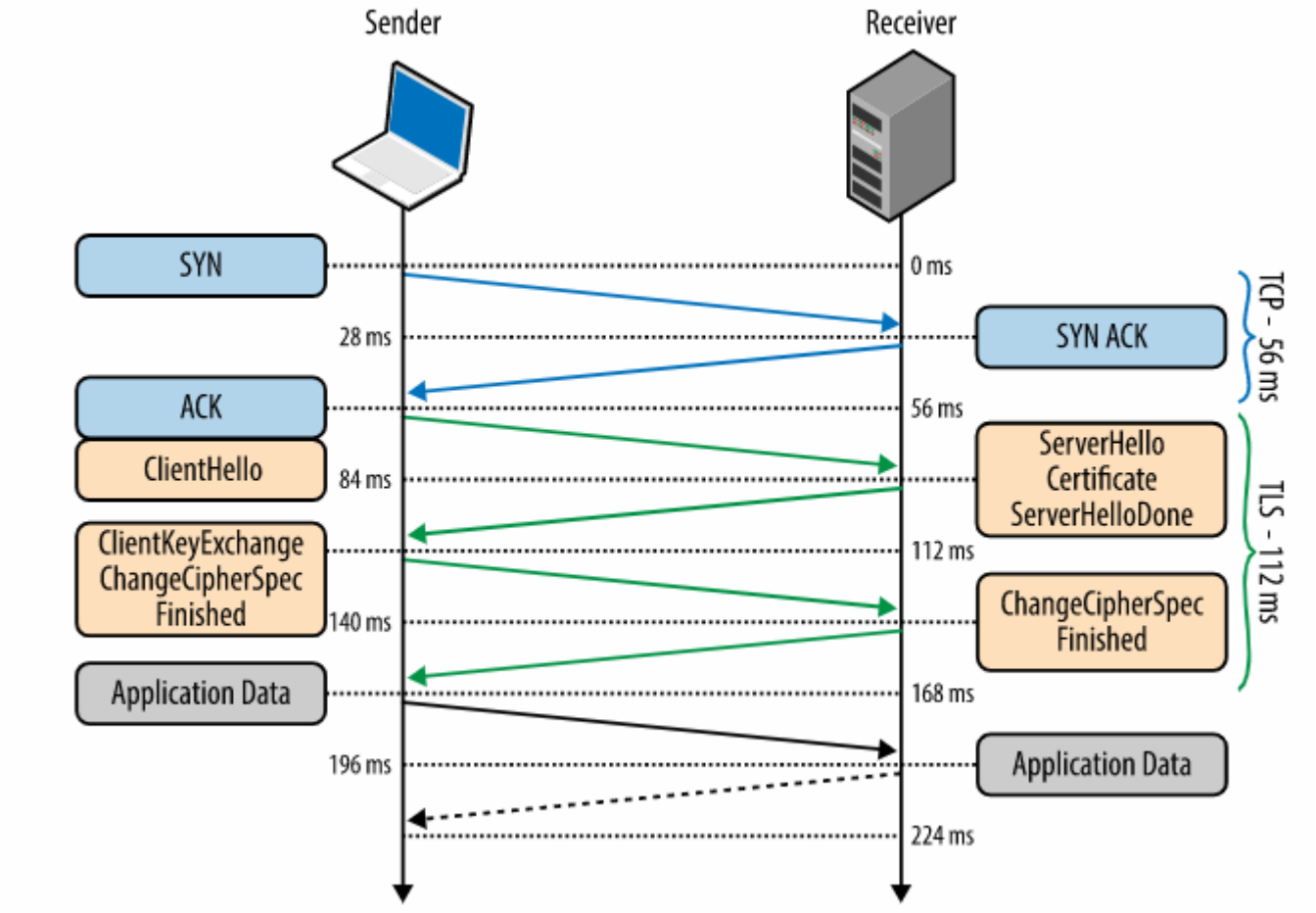
X X X

Overview Heartbleed

- Discovered in 2014
- Exploits a bug in the OpenSSL implementation of the TLS “heartbeat hello” extension
- Can affect both client and server side

X X

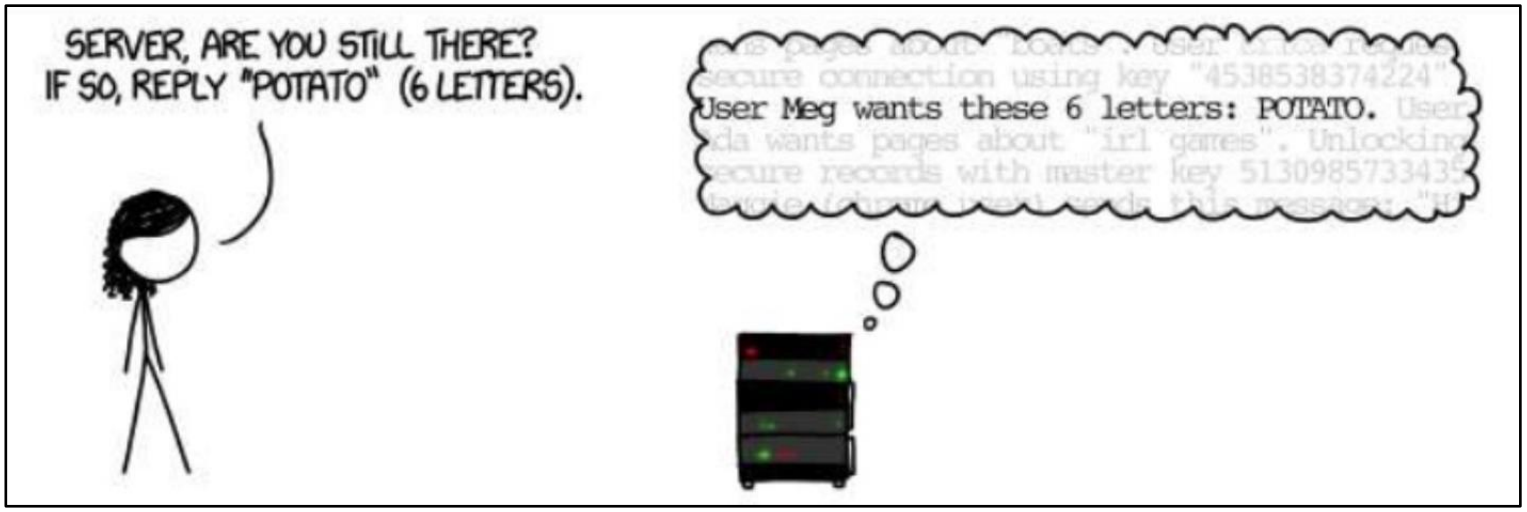
Recap TLS 1.2 Handshake (Server Authentication only)

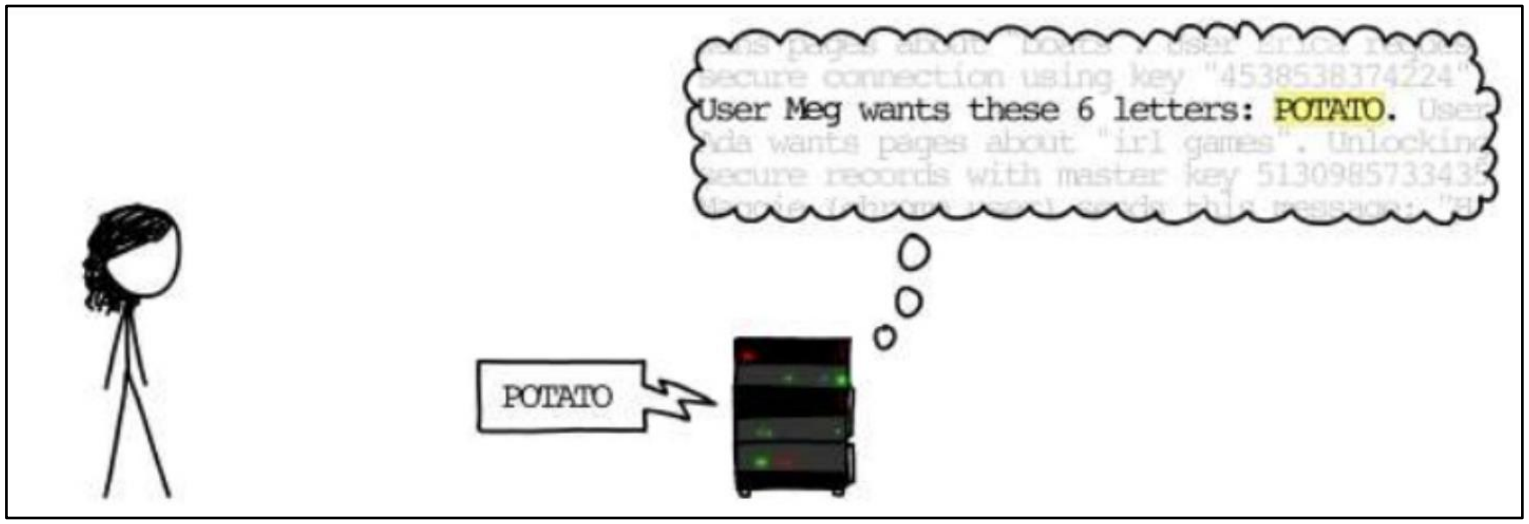


TLS Heartbeat Extension

- Originally TLS had no provisions to keep a client / server connection alive without continuous data transfer
  - Idle connections would timeout instead and a (computationally) expensive handshake (224 ms in the previous example) or a reconnect would have to take place
- The heartbeat extension provides a protocol for "keep-alive" messages that prevent a timeout
  - One endpoint could send out a *HeartbeatRequest* message, which would be immediately responded with a *HeartbeatResponse* message

Heartbeat with incoming Message (correctly) buffered





Heartbeat Request / Response Message

- The Heartbeat protocol messages consist of their type and an arbitrary payload and padding.

heartbeat_request or heartbeat_response

- struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
  } HeartbeatMessage;

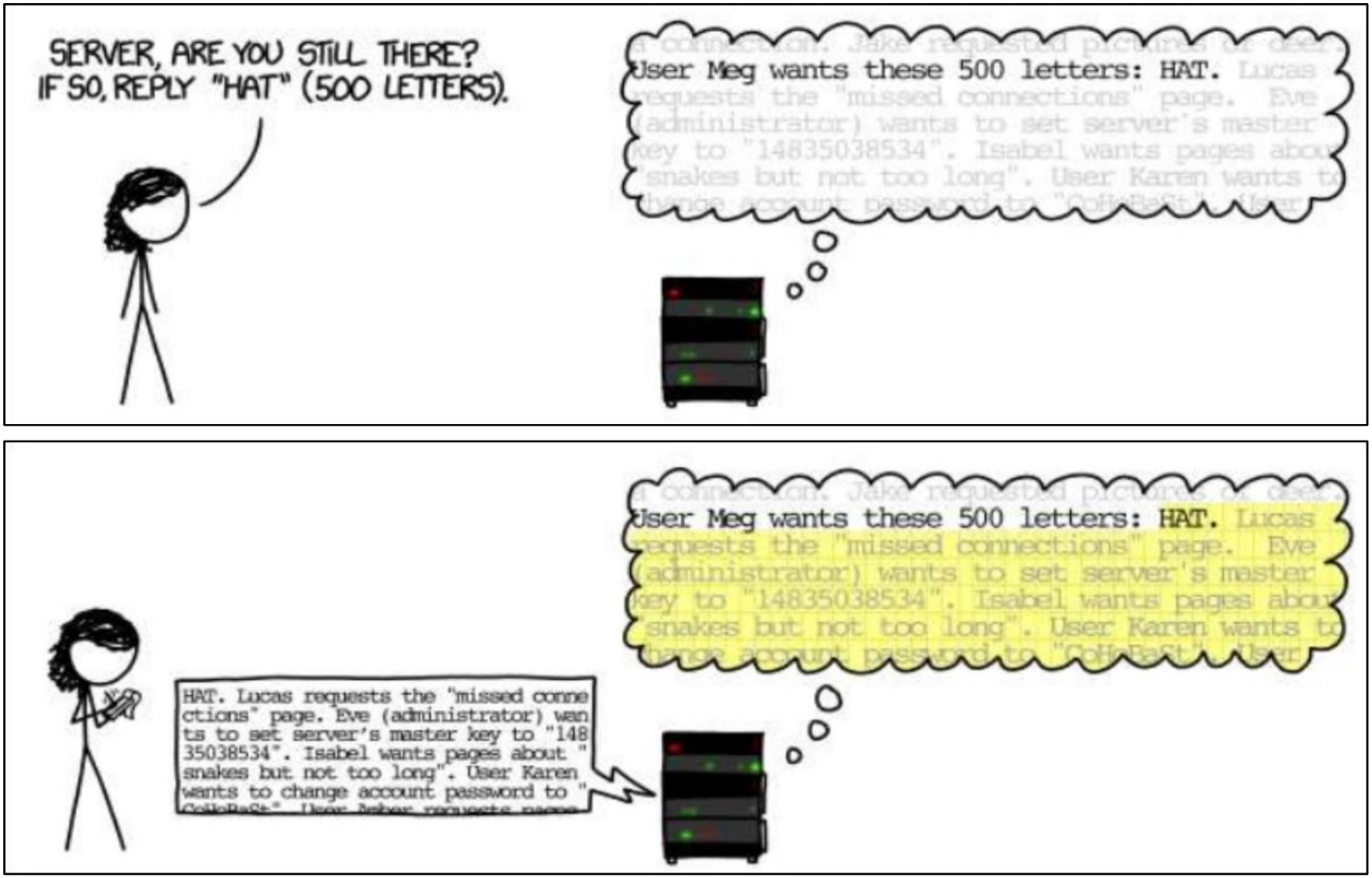
16+ bytes of random content, ignored by receiver

- The sender composes a request message containing a payload with a specified length (i.e. *payload_length*)
- The receiver returns a response message containing a copy of the sender's payload (with length *payload_length*)
- "opaque" is a typdef (i.e., unsigned char)

Heartbleed Exploit

- The server receives a Heartbeat request message and copies it into memory, for further processing
  - However, memory also contains information from other sessions including tokens, keys, session IDs etc.
- If *payload_length* is actually larger than the *payload[..]*, the server will copy memory content beyond the payload array into the response message's payload array (let's call it ret_*payload*), which is then sent back to the sender
- *memcpy(ret_payload, payload, payload_length);*
- Remember, this is C (and not Java or Python), so array boundaries are not checked!
- This is a typical **buffer over read attack**

The Heartbleed Attack



Heartbleed Exploit Extract (Python Code)

- https://gist.github.com/eelsivart/10174134

Heartbleed (CVE-2014-0160) Test & Exploit Python Script

heartbleed.py — Raw

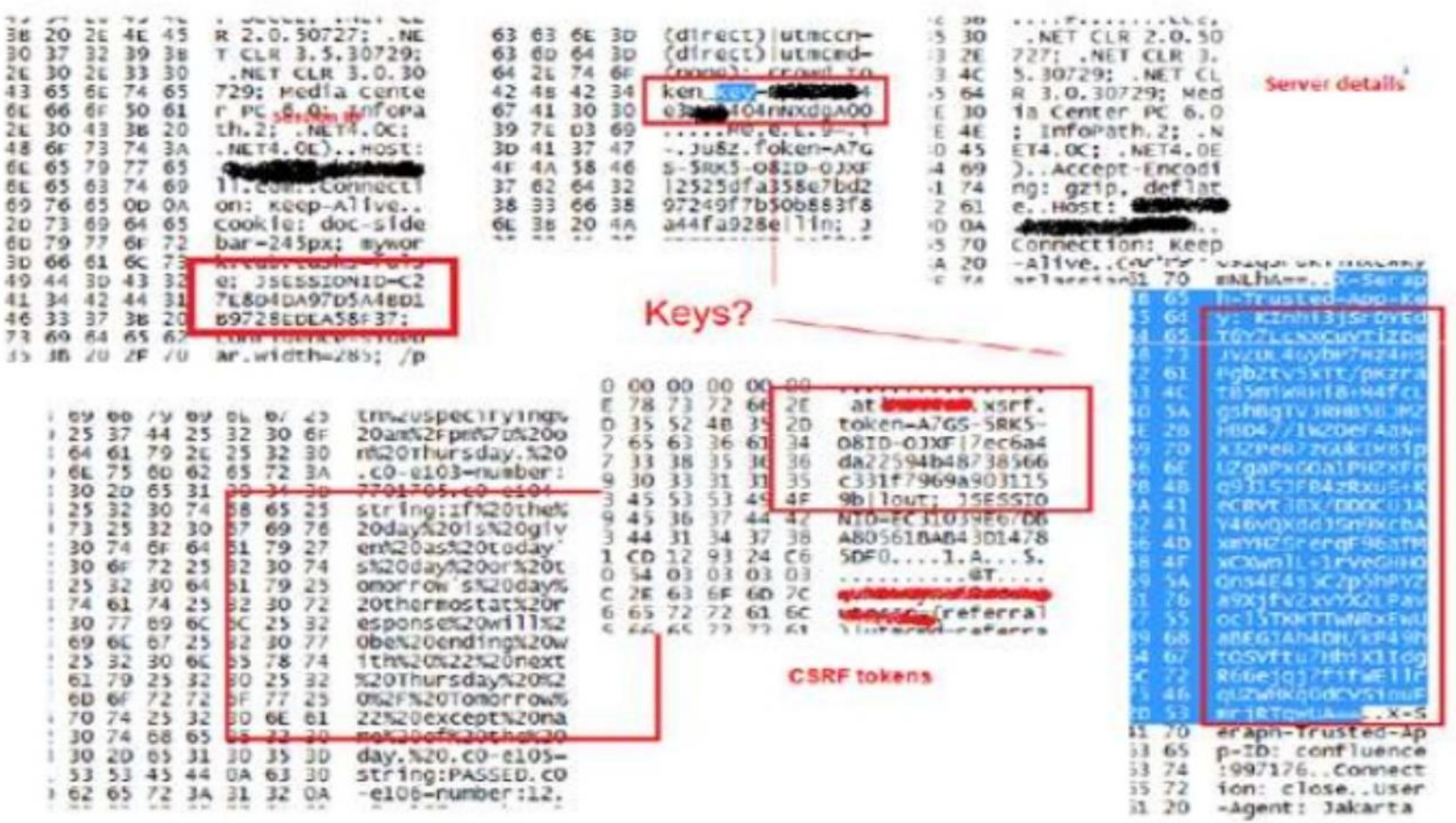
```
#!/usr/bin/python

# Modified by Travis Lee
# Last Updated: 4/21/14
# Version 1.16
#
# -changed output to display text only instead of hexdump and made it easier to read
# -added option to specify number of times to connect to server (to get more data)
# -added option to send STARTTLS command for use with SMTP/POP/IMAP/FTP/etc...
# -added option to specify an input file of multiple hosts, line delimited, with or without a port specified (host:port)
# -added option to have verbose output
# -added capability to automatically check if STARTTLS/STLS/AUTH TLS is supported when smtp/pop/imap/ftp ports are entered and automatically
# -added option for hex output
# -added option to output raw data to a file
# -added option to output ascii data to a file
# -added option to not display returned data on screen (good if doing many iterations and outputting to a file)
# -added tls version auto-detection
# -added an extract rsa private key mode (orig code from epixoip. will exit script when found and enables -d (do not display returned data
#  -requires following modules: gmpy, pyasn1

# Quick and dirty demonstration of CVE-2014-0160 by Jared Stafford (jspenguin@jspenguin.org)
# The author disclaims copyright to this source code.

import sys
import struct
import socket
import time
import select
import re
```

What can be leaked?



What happened next?

- The Heartbleed bug was fixed (of course)
- Further checks and balances were added to validate that payload length was correct

```
struct {
HeartbeatMessageType type;
uint16 payload_length;
opaque payload[HeartbeatMessage.payload_length];
opaque padding[padding_length];
} HeartbeatMessage;
```

==

Heartbleed Impact

- The Heartbleed vulnerability was in all versions of OpenSSL released between March 2012 and April 2014
  - It was a zero-day (i.e., a vulnerability unknown to its owners, developers or anyone capable of mitigating it) **for almost 2 years**
- According to CVE-2014-0160, the following operating system distributions were potentially affected:
  - Debian Wheezy (stable)
  - Ubuntu 12.04.4 LTS
  - CentOS 6.5
  - Fedora 18
  - OpenBSD 5.3
  - FreeBSD 10.0
  - NetBSD 5.0.2
  - OpenSUSE 12.2

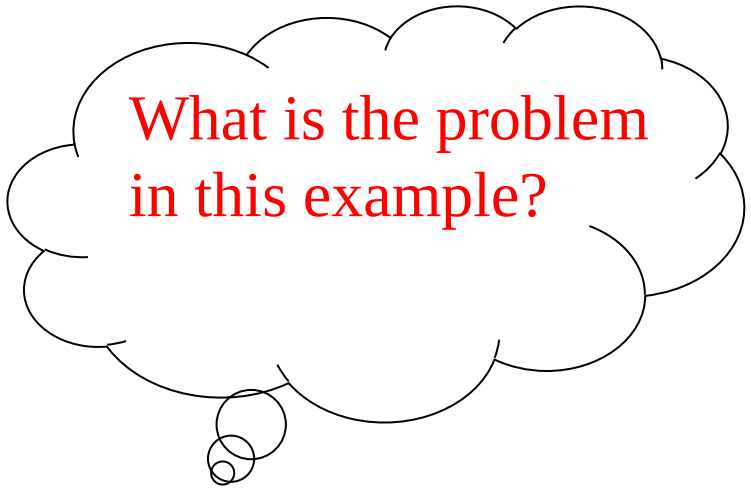
Lessons learnt

- OpenSSL core developer Ben Laurie claimed that a *security audit of OpenSSL would have caught Heartbleed*
- *Some other quotes from the security community:*
  - *"Think about it, OpenSSL only has two fulltime people to write, maintain, test, and review 500,000 lines of business-critical code"*
  - *"The mystery is not that a few overworked volunteers missed this bug; the mystery is why it hasn't happened more often"*
  - *"There should be a continuous effort to simplify the code, because otherwise just adding capabilities will slowly increase the software complexity. The code should be refactored over time to make it simple and clear, not just constantly add new features. The goal should be code that is "obviously right", as opposed to code that is so complicated that "I can't see any problems"*

Related Problem: Buffer Overflow / Stack Overflow

```
#include <string.h>
void foo (char *bar)
{
   char  c[12];
   strcpy(c, bar);
}

int main (int argc, char **argv)
{
   foo(argv[1]);
   return(1);
}
```



X

Example for a Stack Overflow

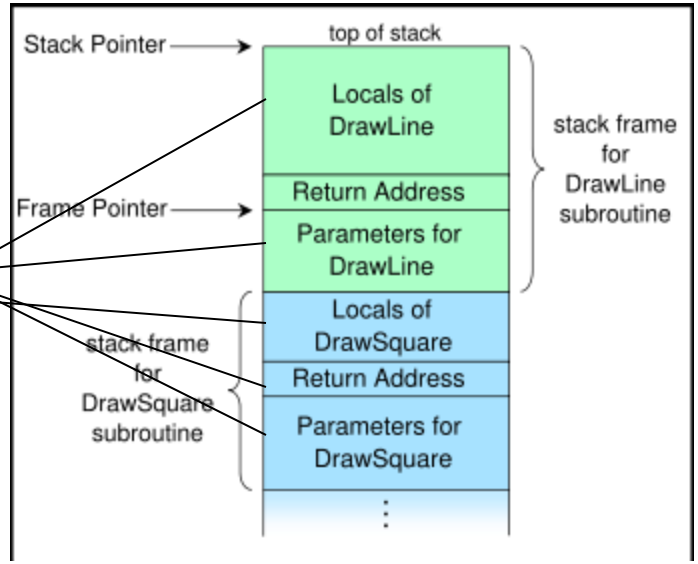
```
#include <string.h>
void foo (char *bar)
{
   char  c[12];
   strcpy(c, bar);
}

int main (int argc, char **argv)
{
   foo(argv[1]);
   return(1);
}
```

- Lets assume the compiled program is called test
- Test is invoked from command line (next slide):
  - "> test hello" will work fine
  - "> test AAAAAAAAAAAAA AAAAAAAA" (> 11 characters) may crash the program

Background Info: The Call Stack

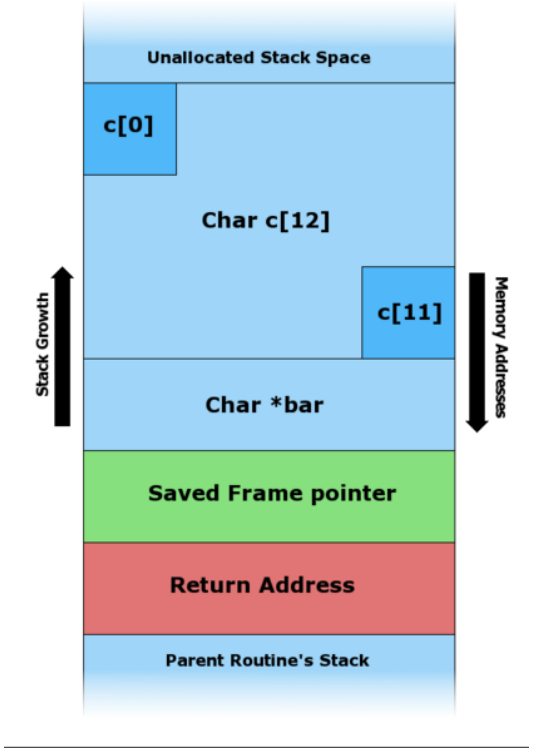
```
main()
{
        DrawSquare(1,1,4,4);
        …
}
…
void DrawSquare(int lux, int luy, int rbx, int rby)
{
        int t1, t2;
        DrawLine(lux, luy, rbx, luy);
        DrawLine(lux, luy, lux, rby);
        …
}
…
void DrawLine(int p1x, int p1y, int p2x, int p2y)
{
        int temp1;
        …
}
```

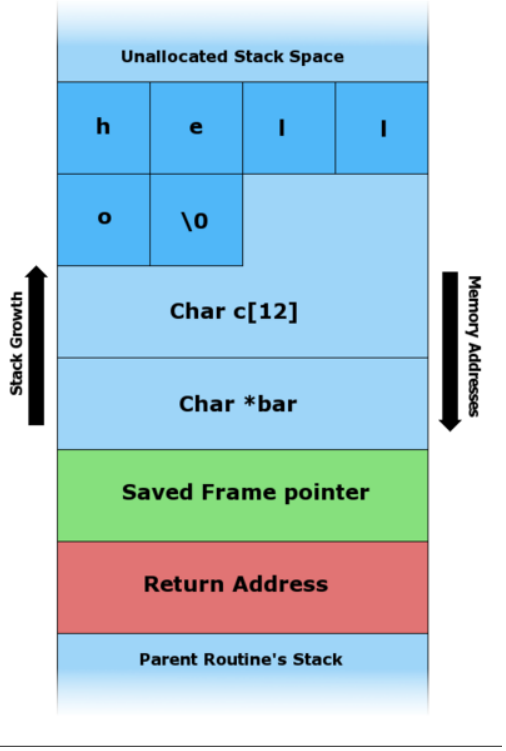


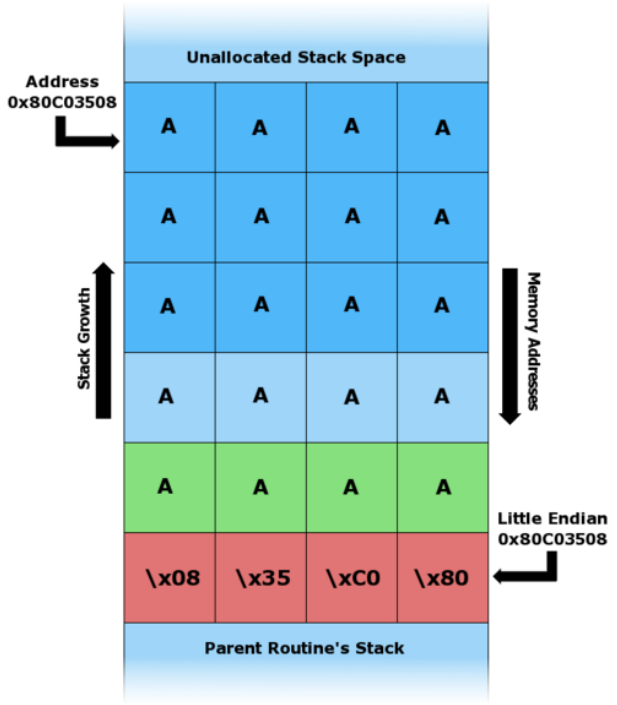
Background Info: The Call Stack

- Each stack frame contains a stack pointer to the top of the frame immediately below
  - The stack pointer is a mutable register
- The stack frame is the collection of all data on the stack associated with one subprogram call. The stack frame generally includes:
  - The return address
  - Argument variables passed on the stack
  - Local variables
- A frame pointer of a given invocation of a function is a copy of the stack pointer as it was before the function was invoked
- If a stack frame is corrupted, i.e. overwritten, arguments, variables and / or return address do change
- If the return address is manipulated, the program can crash, or malware can be executed (with the return address being the start address of the malware im memory)

Example for a Stack Overflow







Buffer Overflow Countermeasures

- Use a programming language that supports automatic bounds checking of buffers
  - Java or Python, but NOT C
- Use a language specific library module that implements info validation in the form of safe buffer handling
- Compilers can produce a warning when an unsafe function call is made, or can add code for buffer overflow detection
- An Operating System can enforce more stringent memory access control so that buffer overflows cannot infringe into the protected areas of the main memory

Buffer Overflow Mitigation using Electric Fence / Boundary Checks

- Here each data object (i.e., array) is guarded by a boundary signature that is checked for its integrity every time that object is accessed
- If the signature has changed as shown below, the data object is deemed to be corrupted, and an alarm will be raised

Before	0xDA	0xEF	Array[0]	Array[1]	…	Array[n]	0xFF	0xED
Attack	0xAA	0xAA	0xAA	0xAA	…	0xAA	0xAA	0xAA

Example Code


```
…
char boundary0 = 0xDA;
char boundary1 = 0xEF
char array[n];
char boundary2 =  0xFF;
char boundary3 = 0xED;
…
// Access array[] only if boundary is intact.
If ((boundary0 == 0xDA) && (boundary1 == 0xEF) && (bounday2 == 0xFF) && (bounday3 == 0xED))
{
   // Access array
  …
}
else
{
  // Error handling
  …
}
```