Dealing with Code That Is Opaque to Static Analysis

Barton P. Miller[†‡], James A. Kupsch[†‡],
Elisa Heymann[†*], Vamshi Basupalli[†‡]

NIST Workshop on Software Measures and Metrics
to Reduce Security Vulnerabilities
Gaithersburg, MD
July 12, 2016

[†] Computer Sciences Department, University of Wisconsin
[‡] DHS Software Assurance Marketplace (SWAMP)
[*] Autonomous University of Barcelona

Recent Experience

- Reviewed high profile vulnerabilities
  - Heartbleed (CVE-2014-010)
  - glibc DNS resolver (CVE-2015-7547)
- Obtained vulnerable source code
- Ran static code analysis tools on each
- Tools failed to find the bugs
- Bug was ***opaque*** to the tools

Heartbleed

At it's heart (sorry), it's just a buffer overflow…

- Failure of the OpenSSL library to validate the heartbeat packet length field (as compared to the size of the actual message).
- Heartbeat packets are contained within TLS packets.
- The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the payload length.
- Since the length field is not checked, `memcpy` can read up to 64KB of memory.

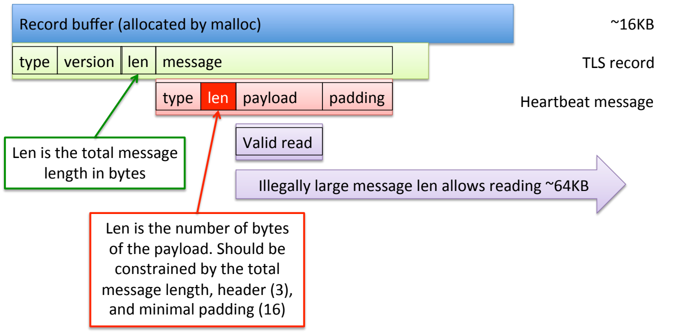
```
memcpy(bp, pl, payload);
```

Destination. Allocated, used, and freed. OK.

Source. Buffer with the heartbeat record. Improperly used.

Length field. Supplied by an untrusted source.

TLS Heartbeat Protocol


Record buffer (allocated by malloc)
~16KB
type
version
len
message
TLS record
type
len
payload
padding
Heartbeat message
Valid read
Illegally large message len allows reading ~64KB
Len is the total message length in bytes
Len is the number of bytes of the payload. Should be constrained by the total message length, header (3), and minimal padding (16)

Heartbleed

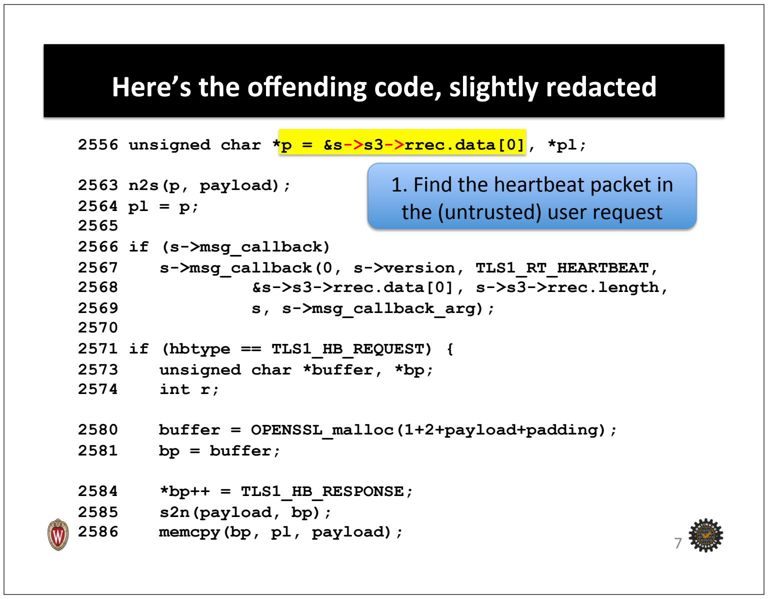
Added length check to remediate:

```
if (1+2+payload+16 > s->s3->rrec.length)
    return 0 // silently discard
```

And none of the current tools could fine the problem…why?

Here's the offending code, slightly redacted

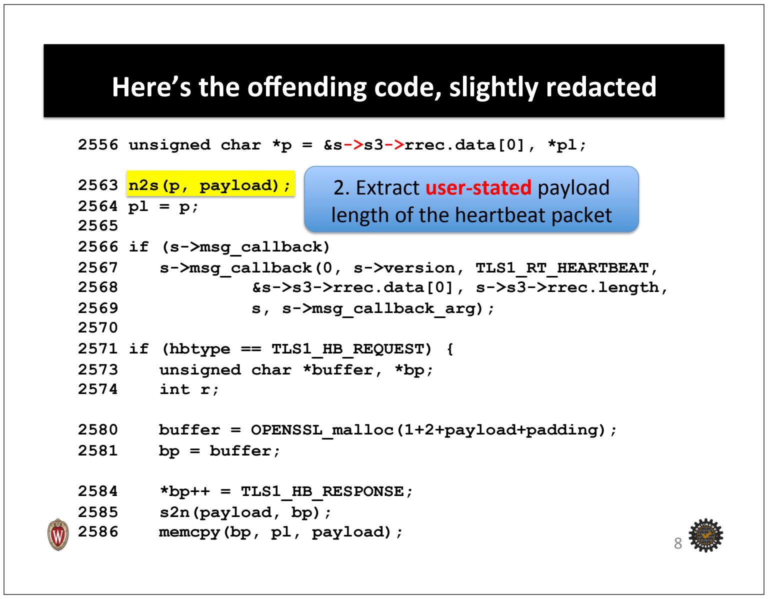
```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
   s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
   unsigned char *buffer, *bp;
   int r;

   buffer = OPENSSL_malloc(1+2+payload+padding);
   bp = buffer;

   *bp++ = TLS1_HB_RESPONSE;
   s2n(payload, bp);
   memcpy(bp, pl, payload);
```



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

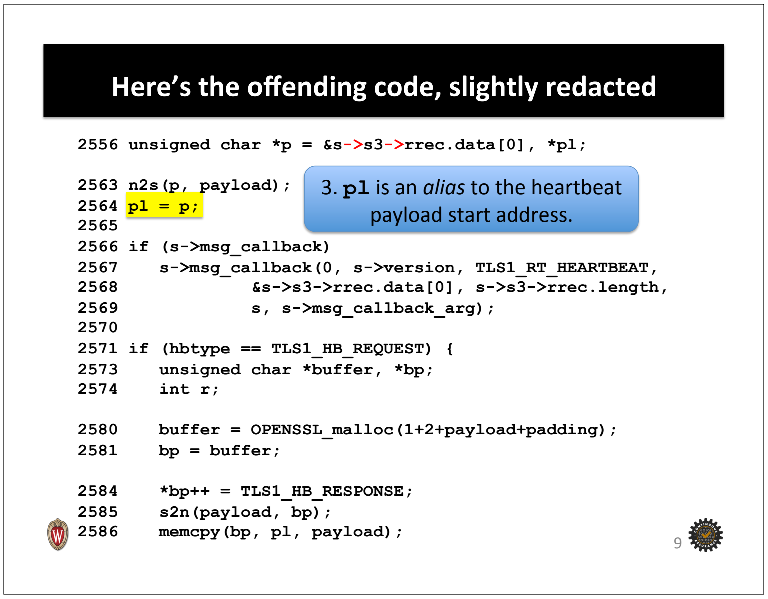
1. Find the heartbeat packet in the (untrusted) user request



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

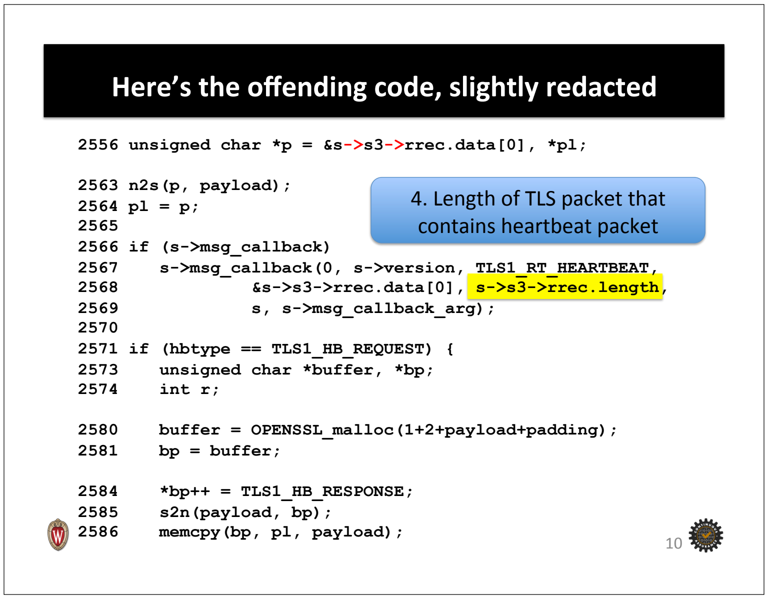
2. Extract **user-stated** payload length of the heartbeat packet



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

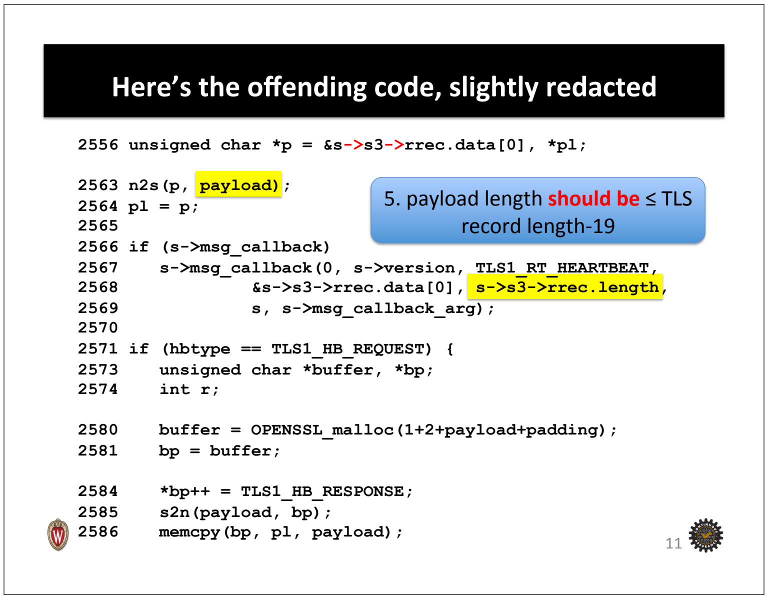
3. **pl** is an *alias* to the heartbeat payload start address.



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

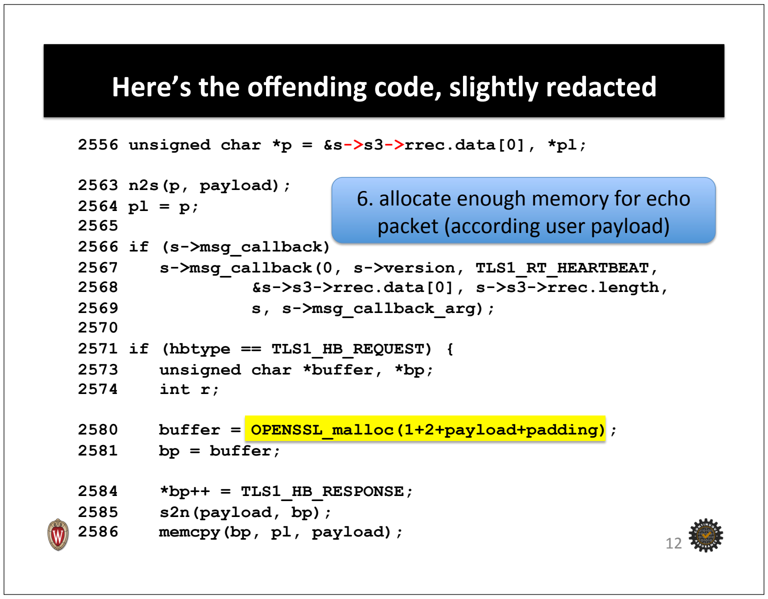
4. Length of TLS packet that contains heartbeat packet



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

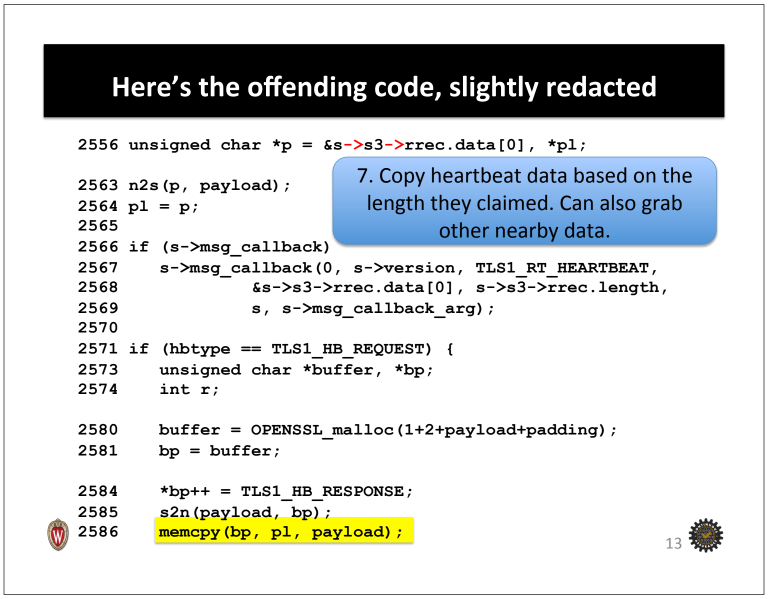
5. payload length **should be** ≤ TLS record length-19



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

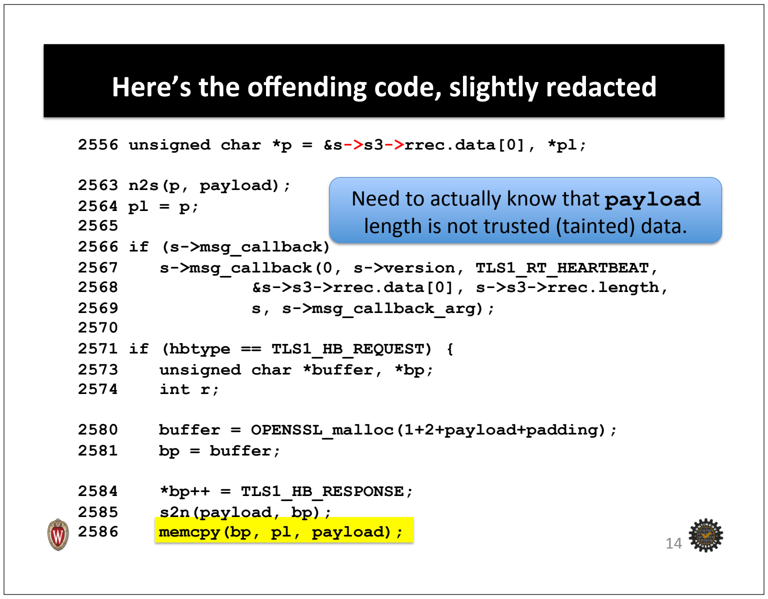
6. allocate enough memory for echo packet (according user payload)



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                    &s->s3->rrec.data[0], s->s3->rrec.length,
                    s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

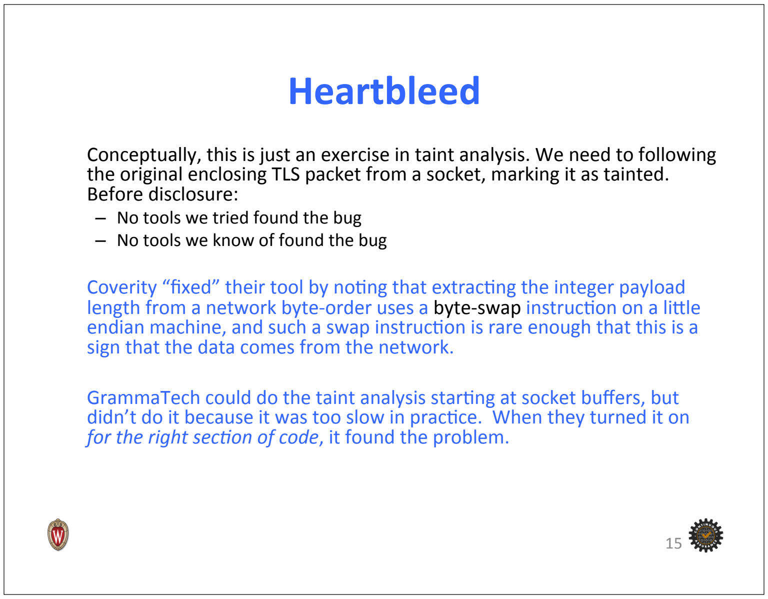
7. Copy heartbeat data based on the length they claimed. Can also grab other nearby data.



Here’s the offending code, slightly redacted

```
unsigned char *p = &s->s3->rrec.data[0], *pl;

n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                    &s->s3->rrec.data[0], s->s3->rrec.length,
                    s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1+2+payload+padding);
    bp = buffer;

    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

Need to actually know that **`payload`** length is not trusted (tainted) data.



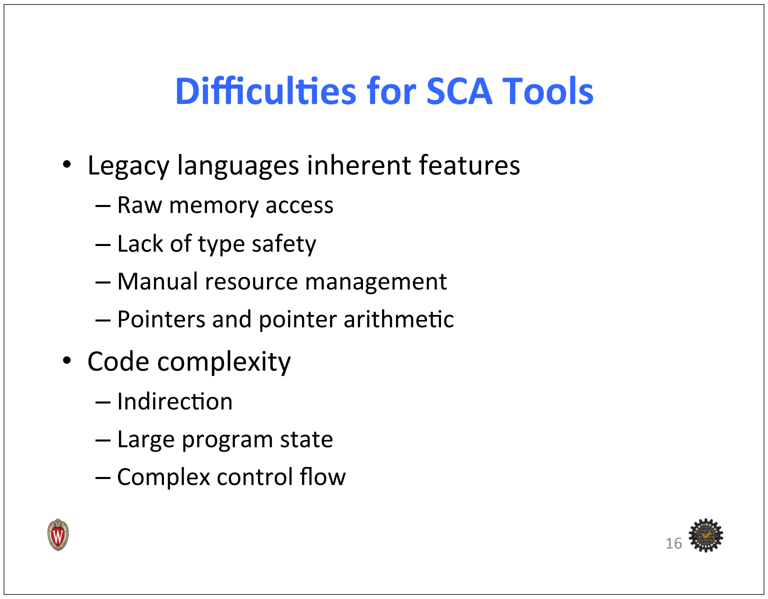
Heartbleed

Conceptually, this is just an exercise in taint analysis. We need to following the original enclosing TLS packet from a socket, marking it as tainted. Before disclosure:

- No tools we tried found the bug
- No tools we know of found the bug

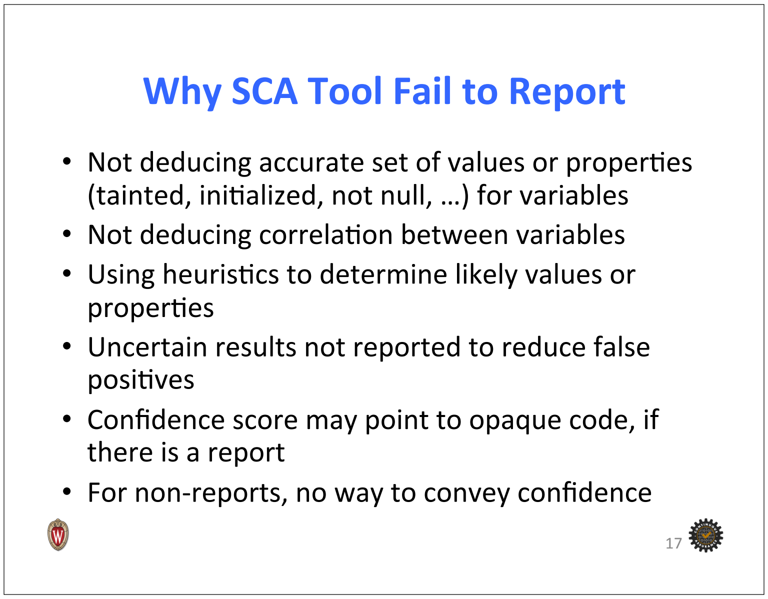
Coverity “fixed” their tool by noting that extracting the integer payload length from a network byte-order uses a byte-swap instruction on a little endian machine, and such a swap instruction is rare enough that this is a sign that the data comes from the network.

GrammaTech could do the taint analysis starting at socket buffers, but didn’t do it because it was too slow in practice. When they turned it on *for the right section of code*, it found the problem.



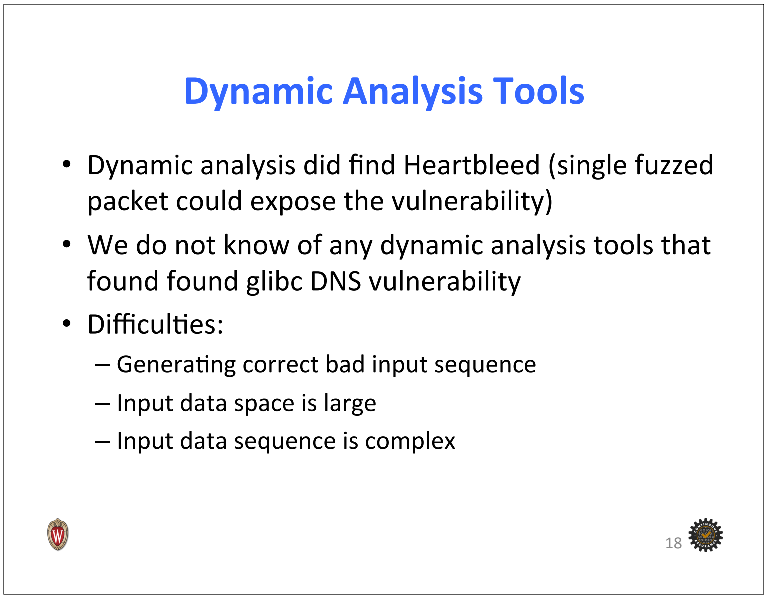
Difficulties for SCA Tools

- Legacy languages inherent features
  - Raw memory access
  - Lack of type safety
  - Manual resource management
  - Pointers and pointer arithmetic
- Code complexity
  - Indirection
  - Large program state
  - Complex control flow



Why SCA Tool Fail to Report

- Not deducing accurate set of values or properties (tainted, initialized, not null, …) for variables
- Not deducing correlation between variables
- Using heuristics to determine likely values or properties
- Uncertain results not reported to reduce false positives
- Confidence score may point to opaque code, if there is a report
- For non-reports, no way to convey confidence



Dynamic Analysis Tools

- Dynamic analysis did find Heartbleed (single fuzzed packet could expose the vulnerability)
- We do not know of any dynamic analysis tools that found found glibc DNS vulnerability
- Difficulties:
  - Generating correct bad input sequence
  - Input data space is large
  - Input data sequence is complex

Goal: Less Opaque Code for SCA

- Two approaches
  - – New code:
    - Use modern languages to prevents some defects
      - – D, Rust, modern C++
    - Use (more) analyzable subset of language
      - – MISRA
      - – Checked C
      - – C++ Core Guidelines, GSL (guideline support library), and SCA
  - – Legacy code (and to a lesser extent new code):
    - Identify parts that are opaque
    - Current metrics do not identify opaque code

Common Metrics

- Metric Types
  - – Simple counts:
    - Lexical elements: lines of code, comments, …
    - Syntactic elements: parameters, types, operators, …
    - Per function, file, or code base
  - – Calculated metrics:
    - Examples: Cyclomatic, Halsted
    - Per function
  - – Relationships between functions, classes, …
    - Examples: Coupling, Cohesion, Connascence
    - Per pair of functions, classes, …
- In our experience, these metrics did not correlate with weaknesses or static analyzability
- Focus: cost to develop, maintain, test, enhance, …

Proposal: Opaqueness Metric

- Develop tools that identify program complexity in terms of opaqueness to analyzability by SCA tools
  - – Semantic complexity of code that reaches a tool's ability to report due to reaching limits of the analysis algorithm's
    - Decidability
    - Implementation
  - – Score regions of the source code with an opaqueness score
  - – Also include rationale for poorly scoring regions
- Provide prescriptive advice to transform the code to be less opaque to SCA (more easily and correctly analyzable)

SCA Tool Providers Path Forward

- Best semantic code analysis is in commercial tools
- SCA tools already have much of the information
  - – Know where assumptions are made
  - – Location of assumptions are accurate
  - – Should be accurate for users of the tool
- Limitations
  - – Inherently not in their interest, reporting limitations is bad for marketing
  - – Specific to the types of problems the tool finds and the power of the tool

Broader Path Forward: Develop Tool

- – Start with existing open source analysis framework
  - Clang Static Analyzer
  - Gcc
- – Fund open source tool based on framework to score the source code based on its opaqueness to static analysis
- – Develop prescriptive guidance on transforming source to make code less opaque

Questions

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

SWAMP
SOFTWARE ASSURANCE MARKETPLACE