Program Analysis of Cryptographic Implementations for Security

Sazzadur Rahaman and Danfeng (Daphne) Yao

Computer Science
Virginia Tech
{sazzad14, danfeng}@cs.vt.edu
Cryptographic Vulnerabilities

Massive Security Bug In OpenSSL Could Affect A Huge Chunk Of The Internet

The 'Heartbleed' security flaw that 

"Unauthorized" memory over read using "memcpy"!

Heartbleed bug still affects thousands of sites

US still has more than 42,000 websites vulnerable to the flaw, which can allow an attacker to steal data directly from websites and users.

More Cryptographic Vulnerabilities

IVs were predictable by the attacker ...

Use of weaker cryptographic schemes (DES, 3DES ...)

Sweet32: Birthday attacks on 64-bit block ciphers in TLS and OpenVPN

Block Ciphers and the Birthday Bound
Other Cryptographic Vulnerabilities

Use of backdoorable cryptographic schemes ...

Leakage of pseudorandom seeds in Juniper Network ...

Secrets are predictable ...
Motivation ...

Real People are getting Hurt!

My Three Years in Identity Theft Hell

Hounded by bill collectors, searched at the airport, thwarted in a house hunt. Here’s how I got free—for now.

By Drew Armstrong

September 13, 2017, 5:01 AM EDT  Updated on September 13, 2017, 8:03 AM EDT
Our Objective

- An empirical study of cryptographic misuse in android applications [Manuel Egele et al. CCS’13]
- Comparing the Usability of Cryptographic APIs [Yasemin Acar et al IEEE S&P’17]

Minimize the gap between the theory and the implementation to help the developer to write more secure code

Can one use program analysis to detect these vulnerabilities?
Cryptographic Program Analysis

Requirement

- Needs to map cryptographic concepts to program analysis mechanisms

Advantages

- Automatically Detect vulnerabilities
- Rich Existing tool-set (data flow analysis, control flow analysis, taint analysis, fuzzing based analysis ...)
- Scalable with reasonable soundness and false positives
Static Program Analysis

begin
int x, y, power;
float z;
input (x, y);
if (y<0)
power = -y;
else
power = y;
z = 1;

CFG

AST

1 → 2 → 4 → n
1 → 4 → 7 → n
1 → 2 → 9 → n
Dynamic Program Analysis

Input Generator

Input → Controlled Environment → Output + Insights

Depends on anomalies in external Behavior …
Our Contribution

- Provide mappings between cryptographic properties and program analysis tools
- Show that static taint analysis can be useful for cryptographic program analysis
- Provide a prototype implementation named “TaintCrypt” using clang-static analyzer
Methodology

Leaked PRNG SEEDs

Should not leak PRNG SEEDs in any form!

Detect the leakage of pseudorandom SEEDs

Used Backdoorable PRNG

Should not use unreliable cryptographic Schemes

Detect the use of unreliable cryptographic Schemes
## Crypto to Program Properties

<table>
<thead>
<tr>
<th>Serial</th>
<th>Rule</th>
<th>Analysis Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Should not use ECB mode in symmetric ciphers *</td>
<td>Taint Analysis</td>
</tr>
<tr>
<td>2</td>
<td>IVs in CBC mode, should be generated randomly *</td>
<td>Taint Analysis</td>
</tr>
<tr>
<td>3</td>
<td>Validity of ciphertexts should not be revealed in symmetric ciphers</td>
<td>Unknown</td>
</tr>
<tr>
<td>4</td>
<td>Validity of ciphertexts should not be revealed in RSA</td>
<td>Unknown</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>Should not have cyclic function calls</td>
<td>Call graph Analysis</td>
</tr>
<tr>
<td>24</td>
<td>Should detect illegal transitions in protocol state machines</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

- 25 security properties
- 20 of them are enforceable using static analysis
- 15 of them are enforceable using static taint analysis

And many more to come …
Taint Analysis

Trusted source

Untrusted source

Sensitive Sink

Trusted source

\[ \downarrow \]

source

tainted

untainted

warning

filter

propagator

sink
Dual Version of Taint Analysis

Insensitive source

Sensitive source

Untrusted Sink

Insensitive source

source

warning

sink

tainted

propagator

filter

untainted
Our Implementation: TaintCrypt

Source code → Scan-build toolchain → TaintCrypt

TaintCrypt* 

configuration

Taintchecker*

LLVM bitcode

Analysis ReportA

* https://github.com/franchiotta/taintchecker
Key Insights for Taint Analysis ...

Source of PRNG seeds are known

Network writer functions are known

Detect the leakage of pseudorandom SEEDs

SEED source

Network writer function

sensitive sources

untrusted sink
Pseudorandom Vulnerability With Taint Analysis

```c
typedef void (*) prng_reseed(void);

prng_reseed = NULL;

void prng_reseed(void) {
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 32;
}

typedef void (*) prng_generate(int is_one_stage);

prng_generate = NULL;

void prng_generate(int is_one_stage) {
    if(!one_stage_rng(is_one_stage))
        prng_reseed();
    for(; prng_output_index <= 31;)
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
}

typedef void (*) print_number(void);

print_number = NULL;

void print_number(void) {
    prng_generate(is_one_stage);
    print_number(prng_temporary, 8);
    ...
}
```

- **memcpy(prng_seed, prng_temporary, 8)**
  - "prng_seed" is propagated to "prng_temporary"
  - Outputs "prng_temporary" to network...

- **prng_reseed()**
  - Uses "prng_temporary" internally...

- **prng_generate(is_one_stage)**
  - void prng_generate(int is_one_stage) {...}

- **print_number(prng_temporary, 8)**
  - void print_number(void) {...}
TaintCrypt in Action

```c
void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng temporary, 32) != 32)
        // Taking false branch
    
    error_handler();
    memcpy(prng_seed, prng

    // Expression 'prng temporary'
    prng_output_index = 8;
    memcpy(prng_key, &prng

    print_number(prng temporary, 32);

    // Untrusted data 'prng temporary' is passed to this sink!
}
```
Heartbleed With Taint Analysis

```c
{
  ...
  hbtype = *p++;
  n2s(p, payload);
  pl = p;
  ...
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);
  ...
}
```

- "payload" is tainted here
- Tainted "payload" is passed to "memcpy"

Diagram:
- Tainted state
- Untainted state
- Filter
- Propagator
- Warning

19
TaintCrypt: Heartbleed...

```c
n2s(p, payload);

1 Within the expansion of n2s(p, payload):

a Expression 'payload'

2 Taking false branch

s->msg_callback;

3 Assuming 'hbtyp

4 Taking true branch

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

    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);

5 Untrusted data 'payload' is passed to this sink!
```
TaintCrypt in Action

```c
RAND_add(data, i, 0);  /* put in the RSA key. */
OPENSSL_assert(enc->iv_len <= (int)sizeof(iv));
if (RAND_bytes(iv, enc->iv_len) <= 0) /* Generate a salt */
    goto err;
/*
 * The 'iv' is used as the iv and as a salt. It is NOT taken
 * the BytesToKey function
 */
if (!EVPBytesToKey(enc, EVP_md5(), iv, kstr, klen, 1, key, NULL)
    Untrusted data 'EVP_md5()' is passed to this sink!
```

21
Limitations of TaintCrypt

- Does not support cross-translational-unit analysis
- Only functions can be used as taint source, sink, propagator and filter
- Soundness vs false positive vs running time trade-off

- Challenging to identify sources and sinks
- Same sources and sinks might not be applicable for multiple projects
Scope of Taint Analysis

Should not use constant cryptographic keys

How can we define source???

Sinks are encrypt or decrypt function

Taint analysis does not help! :(

cipher.encrypt(key, text, iv);

We have to start from the sink to trace back all the sources and see whether any path leads to a constant?

Backward Program Slicing to the rescue!
Scope of Static Analysis

Validity of ciphertexts should not be revealed in RSA!

It is unknown of how to enforce this rule using static analysis! :(

Bleichenbacher Padding Oracle

Daniel Bleichenbacher
Future Work

- Support for cross translational unit analysis
- Automatic sources and sinks discovery
- Support for predicates as filter
- Build similar tool for Java, Android, C#, etc.
- Explore other static analysis mechanisms to cover other rules
[Key Takeways...]

- Static code analysis can be used for cryptographic property enforcements!
- Static taint analysis is very useful for most of the cases.
- Need further research to make cryptographic program analysis more useful for the developer community...
Thanks!