Runtime Verification for Protocol Implementation

Secure Communication in the Quantum Era (SPS G5448) Project Meeting, September 26th, 2019

Christian Colombo
Mark Vella
Steps led by UM

2B - Identify protocol-level security mechanisms
(March 2020 → March 2021)

3B - Deploy implementation-level security mechanisms
(October 2020 → October 2021)
Progress

Identification of protocol-level security mechanisms (2B)

**Identified different level** at which RV can be useful

Design of runtime verification architecture at these various levels (2B)

Including enforcement of a Trusted Domain through RV

Preliminary implementation of the top level (3B)
## Levels of abstraction of security threats

<table>
<thead>
<tr>
<th>(High level) Wrong protocol implementation</th>
<th>The protocol implementation might deviate from the verified (theoretical) design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Medium level threats</strong></td>
<td>Malware, Data leaks, etc</td>
</tr>
<tr>
<td><strong>Low level threats</strong></td>
<td>Arithmetic overflows, undefined downcasts, and invalid pointer references</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>Can hardware be trusted? Side Channel attacks?</td>
</tr>
</tbody>
</table>
Design using RV

Use some specialised hardware to isolate sensitive processes

Place monitors at strategic points
Design of architecture

Commodity h/w and stock OS
- User-mode
  - AGKE code
  - Token calls
- Kernel-mode
  - Kernel trap
  - USB/network drivers

Crypto h/w token
- Crypto OS
- Generate/Retrieve
- APDU
- Keys
- Ciphers
- MCU

AGKE network exchanges

Full isolation

Crypto OS calls
- Plain/ciphertext data exchanges
- NO key transfers

NO external code provisioning
Check for data leaks (medium level)

Check code while executing (low and high level)

Unmodified H/W & OS (Untrusted domain)
- User-mode
- Kernel-mode

Protocol code
- H/W module calls
- Kernel trap
- USB/network drivers

Protocol network exchanges

H/W security module (Trusted domain)

Crypto OS
- Generate/Retrieve
- Keys
- MCU + crypto co-processor

Crypto OS calls
- Plain/ciphertext data exchanges
- NO key transfers

NO runtime code provisioning
Preliminary implementation case study

Elliptic Curve Diffie-Hellman Exchange (ECDHE)
Preliminary implementation

Setup using Binary-level instrumentation
Preliminary implementation

Setup using Binary-level instrumentation

Through which monitors can gain visibility

- Web server
- Binary instrumentation
- Firefox implementation (C code)
- Runtime Verification
Properties verified (High level) on ECDHE

Digital certificate verification is done (in order to authenticate public keys sent by peers)

Validation of remote peer's public key on each exchange is done (unless the session is aborted)

Once master secret is established, private keys should be scrubbed from memory (to limit the impact of memory leak attacks such as Heartbleed, irrespective of whether the session is aborted)
Feasibility study of approach

Is the approach possible for a realistic code base?

Is the approach feasible in terms of overheads?

Used the Firefox case study on top 100 Alexa sites
Feasibility study

Web server

Binary instrumentation

Firefox implementation
(C code)

Runtime Verification

```c
/* TID 0x1003 */
200 ms 0x1003 PR_Close()
200 ms 0x1003 fd:0x7faa3ded6e20
/* TID 0xffb */
312 ms 0xffb SSL_ImportFD()
312 ms 0xffb ret:0x7faa43591940
312 ms 0xffb SSL_AuthCertificateHook()
312 ms 0xffb fd:0x7faa43591940
312 ms 0xffb ret:0x0
312 ms 0xffb PR_Connect()
312 ms 0xffb fd:0x7faa43591940
531 ms 0xffb SECKEY_CreateECPrivateKey()
531 ms 0xffb cx:0x7faa3deda988
532 ms
   | 0xffb EC_VerifyPublicCKey()
532 ms
   | 0xffb ret:0x0
532 ms 0xffb ret:0x7faa3dd66020::7faa3dd66020 c0 fe d9
3d
   ...
533 ms 0xffb SECKEY_CreateECPrivateKey()
533 ms 0xffb cx:0x7faa3deda988
534 ms
   | 0xffb EC_VerifyPublicCKey()
539 ms
   | 0xffb ret:0x0
539 ms 0xffb ret:0x7faa3dd68020::7faa3dd68020 00 f1 10
```
Feasibility study

Challenge: Threads didn’t correspond to sessions
Challenge: efficiency vs precision

How do you keep track which method calls belong to which session?

Firefox is built for efficiency not monitorability

Two options:

Trace all method calls

Change Firefox implementation
Challenge: efficiency vs precision

How do you keep track which method calls belong to which session?

Another option:

Trace only the methods of interest

Use a heuristic (around 98% effectiveness)
What does the specification language look like?

Transitions {
  start → newsession [sslimport]
  newsession → server_connect [prconnect]
  server_connect → failed_cert_auth [sslauthcertcompl]
  failed_cert_auth → close [prclose\mcParent=mc;]
  close → certerr_ok [destroypk\mc.hasParent(mcParent)]
}

failed_cert_auth → certerr_bad [eot]
  close → certerr_bad [eot]
Overheads measurement

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Pages</th>
<th>Page load time (ms)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mean</td>
<td>std. dev.</td>
<td></td>
</tr>
<tr>
<td>No RV</td>
<td>1,000</td>
<td>6,918.37</td>
<td>24,870.86</td>
<td></td>
</tr>
<tr>
<td>With RV</td>
<td>1,000</td>
<td>7,282.35</td>
<td>27,328.9</td>
<td></td>
</tr>
<tr>
<td>Mean overhead</td>
<td></td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilcoxon signed-rank test</td>
<td></td>
<td>p=0.281</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overheads measurement

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Pages</th>
<th>Page load (ms)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mean</td>
<td>std. dev.</td>
<td></td>
</tr>
<tr>
<td>No RV</td>
<td>1,000</td>
<td>6,918.37</td>
<td>24,870.86</td>
<td></td>
</tr>
<tr>
<td>With RV</td>
<td>1,000</td>
<td>7,282.35</td>
<td>27,328.9</td>
<td></td>
</tr>
<tr>
<td>Mean overhead</td>
<td></td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilcoxon signed-rank test</td>
<td></td>
<td>p=0.281</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.05 ms per page
Lessons learnt

Good start with promising results - approach seems feasible

Beware:

**Program comprehension is required**, both for setting up function hooks as well as to enable individual TLS session monitoring

Real-world code tends to be written in a manner to favor efficient execution rather than monitorability (eg, was difficult to keep track of particular sessions on the server)
Moving forward
Implementation on SEcube Development Kit

Key generation will take place on dedicated HW

While still monitoring the protocol execution
Design of architecture

Unmodified H/W & OS (Untrusted domain)

User-mode

Protocol code

H/W module calls

Kernel trap

Kernel-mode

USB/network drivers

Protocol network exchanges

Crypto OS calls
Plain/ciphertex data exchanges
NO key transfers

NO runtime code provisioning

RV + Binary-level function call tracing

RV + Binary-level taint inference
Plan

Identification of (the actual) protocol-level properties (D1) deadline Dec 2019

Implementation

Setup with SEcube hardware (next step with Peter)

Monitoring our “quantum” protocol with this setup

Low level runtime verification (using existing libraries)

Taint inference