redhat.

Dealing with cache based attacks in cryptography

Speculating on cache attacks

Simo Sorce
Sr. Principal Software Engineer
2019/02/19

What are cache based attacks ?

In cryptography

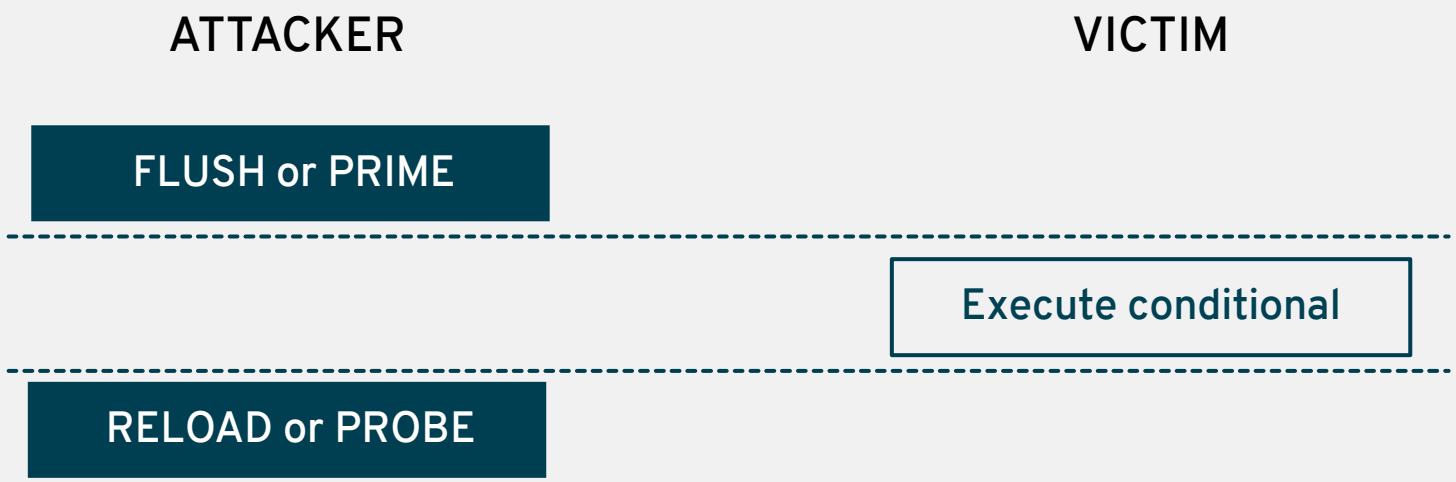
In cryptographic operations, leaking the internal state of a computation can lead directly to key compromises or, less dramatically, to the ability to create forgeries. Cache based attacks can be used to:

- Detect input dependent branch points
- Detect table lookups
- Detect error conditions

All of these events can lead to knowledge about the outcome or path taken by an internal operation, in a ways similar to **timing** based attacks.

Classic Example

Can be executed against data or instruction caches depending on what is more convenient

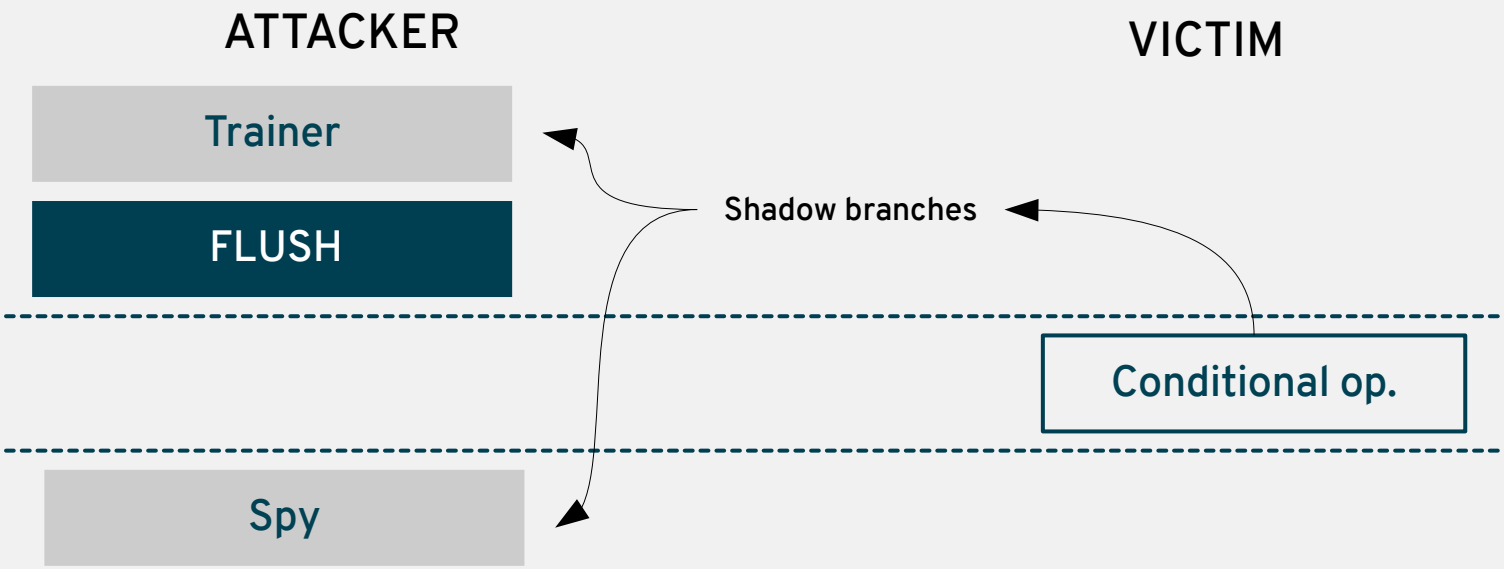


Use timing on reload to figure out what has been cached by the CPU while executing the victim's code, infer internal status.

This simple attack may be too coarse in some cases.

Speculation aided cache attack

Using Branch Predictor (should be mitigated by IBPB/STIBP)



Train the branch predictor then flush your own caches for the “spy” branch. Let the victim code run, then run your own “spy” branch. See if the “trainer offset” flushed line was loaded by the “spy” (measure time to access).
If **not,** the victim branch fired and retrained the BP.

How do we handle these attacks ?

In cryptographic libraries

The only way is to avoid conditionals like the plague, even where previously they would have been undetectable via classic timing attacks.

- All computations must be time and memory access constant
- Remove table lookups
- Compute even after errors, and report errors in silent ways

All of these tend to make code slower.

Historically only timing attacks were considered likely, however since then running “untrusted” code on shared hosts has become a lot more common (VPSs, PaaSs, containers). Not all cryptographic libraries developers feel ready to fight local attacks, considered way too hard to defeat.

Example


```
memcpy(message, terminator + 1, message_length);
*length = message_length;
```

```
/* fill destination buffer fully regardless of outcome. Copies the message
 * in a memory access independent way. The destination message buffer will
 * be clobbered past the message length. */
shift = padded_message_length - buflen;
cnd_memcpy(ok, message, padded_message + shift, buflen);
offset -= shift;
/* In this loop, the bits of the 'offset' variable are used as shifting
 * conditions, starting from the least significant bit. The end result is
 * that the buffer is shifted left exactly 'offset' bytes. */
for (shift = 1; shift < buflen; shift <<= 1, offset >>= 1)
  {
    /* 'ok' is both a least significant bit mask and a condition */
    cnd_memcpy(offset & ok, message, message + shift, buflen - shift);
  }

/* update length only if we succeeded, otherwise leave unchanged */
*length = (msglen & (-(size_t) ok)) + (*length & ((size_t) ok - 1));
```

x3 - x5

Do we continue like this ?

Considering that:

- It is very hard to reason in this way
- Compilers tend to optimize away safeguards
- Some “safer” higher level languages do not have a level of control that allows to deal with these issues in this way

New instructions to protect critical sections by messing with caches ?

THANK YOU


plus.google.com/+RedHat


facebook.com/redhatinc


linkedin.com/company/red-hat


twitter.com/RedHat


youtube.com/user/RedHatVideos