PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild

Daniel De Almeida Braga
Pierre-Alain Fouque
Mohamed Sabt
CCS’21 - November 2021
Context and Motivations
What to expect from a PAKE, starting from a password:

- Authentication
- End up with strong key
- Resist to (offline) dictionary attack

Lots of different PAKEs (two main families: balanced - asymmetric).
Why Looking at PAKEs?

Recent interest:

- Wide Deployment of Dragonfly in WPA3
- CFRG competition for new standard (OPAQUE and CPace)
Why Looking at PAKEs?

Recent interest:

- Wide Deployment of Dragonfly in WPA3
- CFRG competition for new standard (OPAQUE and CPace)

Practical security considerations

- Dragonfly and WPA3: Dragonblood\(^1\) and attack refinement\(^2\)
- Partitioning Oracle Attack\(^3\) applied to some OPAQUE implementations

Case study: Secure Remote Password

---

\(^1\) M.Vanhoef and E.Ronen *Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd*. In IEEE S&P. 2020


Available for a long time => de facto standard for more than 20 years

What about SRP implementations in the wild?
Available for a long time => de facto standard for more than 20 years

What about SRP implementations in the wild?

• Recent work on SRP at ACNS\(^1\)

---

\(^1\) A.Russon Threat for the Secure Remote Password Protocol and a Leak in Apple’s Cryptographic Library. In ACNS. 2021
SRP Protocol Overview

Client

\[ a = \text{rand}(1, p-1) \]
\[ A = g^a \mod p \]

\[ \text{user_id}, A \]

\[ \text{salt}, B \]

\[ u = H(A||B) \]
\[ x = H(\text{salt}||H(\text{user_id}||\text{pwd})) \]
\[ S = (B-kg^x)^{a-u} \mod p \]
\[ K = H(S) \]

client_verify

server_verify

Verify server

Server

\[ \text{salt}, v = \text{lookUp(user_id)} \]
\[ b = \text{rand}(1, p-1) \]
\[ B = kv + g^b \mod p \]

\[ u = H(A||B) \]
\[ S = (Av^u)^b \mod p \]
\[ K = H(S) \]

Verify client
SRP Protocol Overview

Client:
- $a = \text{rand}(1, p-1)$
- $A = g^a \mod p$
- $u = H(A || B)$
- $x = H(\text{salt} || H(\text{id}:\text{pwd}))$
- $S = (B - kg^x)^a \cdot u^x \mod p$
- $K = H(S)$
- client_verify

Server:
- $\text{salt, } v = \text{lookUp(user_id)}$
- $b = \text{rand}(1, p-1)$
- $B = kv + g^b \mod p$
- $u = H(A || B)$
- $S = (Av^b)^a \mod p$
- $K = H(S)$
- server_verify

$x = H(\text{salt}, H(\text{id}:\text{pwd}))$
$v = g^x \mod p$
SRP Protocol Overview

Client

- $a = \text{rand}(1, p-1)$
- $A = g^a \mod p$

Server

- $x = H(\text{salt}, H(\text{id}:pwd))$
- $v = g^x \mod p$

Commit

- $b = \text{rand}(1, p-1)$
- $B = kv + g^b \mod p$

Verification

- $u = H(A||B)$
- $x = H(\text{salt}||H(\text{user_id}:pwd))$
- $S = (B-kg^x)^a \cdotux \mod p$
- $K = H(S)$

Client Verify

- $u = H(A||B)$
- $S = (Av^u)^b \mod p$
- $K = H(S)$

Server Verify

- $x = H(\text{salt}, H(\text{id}:pwd))$
- $v = g^x \mod p$
SRP Protocol Overview

Client

\[ a = \text{rand}(1, p-1) \]
\[ A = g^a \mod p \]

Server

\[ x = H(\text{salt}, H(\text{id}: \text{pwd})) \]
\[ v = g^x \mod p \]

user_id, A

\[ B = \text{rand}(1, p-1) \]
\[ v = \text{lookUp}(\text{user_id}) \]
\[ B = kv + g^b \mod p \]

Commit

\[ u = H(\text{A}||B) \]
\[ x = H(\text{salt} || H(\text{user_id}: \text{pwd})) \]
\[ S = (B-kg^a)^x \mod p \]
\[ K = H(S) \]

Verification

Verify server

clientVerify

\[ S = (Av^u)^b \mod p \]
\[ K = H(S) \]

serverVerify

Verify client

Verify server
Contributions
Contributions

1. Study of various SRP implementations
2. Highlight a leakage in the root library used for big number arithmetic (OpenSSL)
3. Design PoCs of an offline dictionary attack recovering the password on impacted projects
4. Outline the importance of SCA, especially for PAKEs
Our Main Result

A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.
Our Main Result

A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.

Our Main Result

Flush+Reload\(^1\) and PDA\(^2\)

A **cache-attack** that lets us extract information during OpenSSL the **modular exponentiation** allowing to **recover the password** in a **single measure**.

---

2. T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016
Our Main Result

A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.

Our Main Result

A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.

The Vulnerability
Modular exponentiation in OpenSSL

\[ BN_{\text{mod\_exp}} \]
Modular exponentiation in OpenSSL

$BN\_mod\_exp$
Modular exponentiation in OpenSSL

BN_mod_exp

BN_mod_exp_mont_word
Modular exponentiation in OpenSSL

BN_mod_exp

BN_mod_exp_mont_word

BN_mod_exp_mont_consttime

Classic SWE\textsuperscript{1,2}

---

\textsuperscript{1} C. Percival Cache missing for fun and profit. 2005

\textsuperscript{2} C. Peraida Garia et al. Certified Side Channels. In USENIX Security. 2020
Modular exponentiation in OpenSSL

1 C. Percival Cache missing for fun and profit. 2005
Optimized Square-and-Multiply

\[ \text{bin}(e) = 11010 \ldots \]

\[ \text{res} = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```

Optimized Square-and-Multiply

\[ bin(e) = 11010\ldots \]

\[ res = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
        next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
        next_w = g
        w = next_w
    ...
```
$\text{bin}(e) = 11010\ldots$

$\text{res} = g^e \mod p$

$w$ is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
            w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w * g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
                w = next_w
    ...
```
Optimized Square-and-Multiply

\[ \text{bin}(e) = 11010 \ldots \]

\[ \text{res} = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g
    res = BN_to_mont_word(w)
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w//w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w//g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```
**Optimized Square-and-Multiply**

\[ bin(e) = 1 \ 1 \ 0 \ 1 \ 0 \ldots \]

\[ res = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
        w = next_w
    ...
```

"w is a processor word (e.g. 64 bits)"
\textit{Optimized Square-and-Multiply}

\[
\text{bin}(e) = 1 \ 1 \ 0 \ 1 \ 0 \ldots
\]

\textit{res} = g^e \mod p

\textit{w} is a processor word (e.g. 64 bits)

\begin{verbatim}
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
        next_w = g
        w = next_w
    ...
\end{verbatim}
Optimized Square-and-Multiply

\[ \text{bin}(e) = 1 \ 1 \ 0 \ 1 \ 0 \ldots \]

\[ \text{res} = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w / w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w / g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```
\[ bin(e) = 1 \ 1 \ 0 \ 1 \ 0 \ldots \]

\[ res = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```

...
Optimized Square-and-Multiply

\[ \text{bin}(e) = 11010 \ldots \]

\[ \text{res} = g^e \mod p \]

\( w \) is a processor word (e.g. 64 bits)

---

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```

---
Exploiting the Leakage
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g # uint64_t
    res = BN_to_mont_word(w) # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...

```python
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```
def BN_mod_exp_mot_word(g, w, p):
...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
...

def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w / w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w / g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...

Trace Acquisition
def BN_mod_exp_mot_word(g, w, p):
    ...
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
            w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
        next_w = g
        w = next_w
    ...

```python
def BN_mod_exp_mot_word(g, w, p):
    
    w = g  # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w * w
        if next_w/w != w:
            w = next_w
            res = BN_mod_mul(res, w, p)
    next_w = 1
    w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
        next_w = w * g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
    next_w = g
    w = next_w
    ...
Trace Interpretation

Long pause = overflow

Cycles needed to reload

Time

250 300 350
Rules ($b \in \{0,1\}$):

- $Vvv$ -> $111b$
- $Vvvv$ -> $yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$
- $Vv\ldots v$ -> $0 \ldots 0yyyyb$
Trace Interpretation

Long pause = overflow

Rules ($b \in \{0,1\}$):

- $Vvvv \rightarrow 111b$
- $Vvvv \rightarrow yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$
- $Vv.....v \rightarrow 0 ... 0yyyyb$

$Vvv \ Vvvv \ Vvvv \ Vvvv \ Vvvv \ Vvvv \ Vvvv \ Vvvv$
Trace Interpretation

Rules ($b \in \{0,1\}$):

- $Vvvv \to 111b$
- $Vvvv \to yyyyb, yyyy \in \{110b, 10bb, 0111\}$
- $Vv....v \to 0 ... 0yyyyb$

$Vvv Vvvv Vvvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv 111b$
Trace Interpretation

Rules ($b \in \{0,1\}$):
- $Vvvv \rightarrow 111b$
- $Vvvv \rightarrow yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$
- $Vv......v \rightarrow 0 ... 0yyyyb$

$Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv 111b yyyyb$
Trace Interpretation

Rules ($b \in \{0,1\}$):

- $Vvvv \rightarrow 111b$
- $Vvvv \rightarrow yyyb, yyy \in \{110b, 10bb, 0111\}$
- $Vv.....v \rightarrow 0 \ldots 0yyyb$

$Vvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv Vvvv 111b yyyb 0yyyyb yyyyb yyyyb yyyyb bbbb$
Dictionary Attack

Client: \( x = H(salt \| H(user\_id:password)) \)

\( v = g^x \mod p \)
Dictionary Attack

Client: \( x = H(\text{salt} \mathbin{|} | H(\text{user\_id:password})) \)
\( v = g^x \pmod{p} \)

\( b \in \{0, 1\} \)
\( yyyy \in \{110b, 10bb, 0111\} \)

Recovered: 1 1 1 b y y y b 0 y y y b 1 1 1 b 0 y y y b
Dictionary Attack

Client: \( x = H(\text{salt} || H(\text{user_id}:\text{password})) \)
\( v = g^x \mod p \)

\( b \in \{0,1\} \)
\( yyy \in \{110b, 10bb, 0111\} \)

<table>
<thead>
<tr>
<th>Recovered:</th>
<th>1 1 1 b y y y y b 0 y y y y b 1 1 1 b 0 y y y y b</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwd_1</td>
<td>1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>pwd_2</td>
<td>1 1 0 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1</td>
</tr>
<tr>
<td>pwd_3</td>
<td>0 1 1 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>pwd_4</td>
<td>1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>pwd_5</td>
<td>0 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>pwd_n</td>
<td>1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Password</th>
<th>X value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dictionary Attack

Client: \( x = H(\text{salt} \ || \ H(\text{user_id}:\text{password})) \)

\[ v = g^x \mod p \]

Recovered:

<table>
<thead>
<tr>
<th>Password</th>
<th>X value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwd_1</code></td>
<td>1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td><code>pwd_2</code></td>
<td>1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1</td>
</tr>
<tr>
<td><code>pwd_3</code></td>
<td>0 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td><code>pwd_4</code></td>
<td>1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td><code>pwd_5</code></td>
<td>0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
</tr>
<tr>
<td><code>pwd_n</code></td>
<td>1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

\( b \in \{0,1\} \)

\( yyyyy \in \{110b, 10bb, 0111\} \)
Dictionary Attack

Client: $x = H(salt \ || \ H(user_id:password))$
$v = g^x \mod p$

$b \in \{0,1\}$
$\{110b, 10bb, 0111\}$

<table>
<thead>
<tr>
<th>Recovered:</th>
<th>1 1 1 b y y y b 0 y y y b 1 1 1 b 0 y y y b</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwd_1</td>
<td>1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>pwd_2</td>
<td>1 1 0 0 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1</td>
</tr>
<tr>
<td>pwd_3</td>
<td>0 1 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>pwd_4</td>
<td>1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1</td>
</tr>
<tr>
<td>pwd_5</td>
<td>0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>pwd_n</td>
<td>1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Password</th>
<th>X value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dictionary Attack

Client: \( x = H(\text{salt} \mid \mid H(\text{user_id}:\text{password})) \)
\[
v = g^x \mod p
\]

Recovered:

|(pwd_1) | 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 1 |
|(pwd_2) | 1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1 |
|pwd_3  | 0 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 |
|pwd_4  | 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 |
|pwd_5  | 0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0 1 |
...     |                                                |
|pwd_n  | 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1 12 |

<table>
<thead>
<tr>
<th>Password</th>
<th>X value</th>
<th>Diff score</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwd_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwd_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwd_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwd_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwd_5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...     |        |           |
|pwd_n  |        |           |

\( b \in \{0, 1\} \)
\( yyyyy \in \{110b, 10bb, 0111\} \)
Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
  - $k$ bits of information => $2^{-k}$ probability of false positive/negative
Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
  - $k$ bits of information $\Rightarrow 2^{-k}$ probability of false positive/negative

For a $n$ bits exponent, we get $k = 0.4n + 2$ bits on average (verified empirically)
Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
  - $k$ bits of information $\Rightarrow 2^{-k}$ probability of false positive/negative

For a $n$ bits exponent, we get $k = 0.4n + 2$ bits on average (verified empirically)

SHA-1: 66 bits of information
SHA-256: 104 bits of information
• Unprivileged spyware on the victim station

• Victim tries to connect

• MitM can help to gather more information (optional)
Classical Workflow
Classical Workflow

SRP Protocol

Victim

Spy process

Trace parsing
Classical Workflow

SRP Protocol

Victim

Spy process

Leaked information

Trace parsing
Classical Workflow

SRP Protocol

Victim

Spy process

Leaked information

Trace parsing

Offline dictionary attack
Classical Workflow

SRP Protocol

Victim
Spy process

Leaked information

Offline dictionary attack

Trace parsing

Remaining passwords
Practical Impact
• Lots of project using OpenSSL are impacted, including
  • OpenSSL TLS-SRP
  • Apple HomeKit ADK
  • PySRP (used in ProtonMail python client)
  • ...
Lots of projects using OpenSSL are impacted, including:

- OpenSSL TLS-SRP
- Apple HomeKit ADK
- PySRP (used in ProtonMail python client)
-...

Wait, how are big numbers managed in high level languages?...
Many reference libraries are based on OpenSSL to manage bignums

They usually (never ?) manage the flag properly

- Ruby/openssl
- Javascript node-bignum
- Erlang OTP

All SRP implementations using these packages / libraries would be affected!
Mitigations & Conclusion
Two choices:

- Patch this particular issue by adding the proper flag
  - Most projects use the bignum API, not the whole SRP
  - Difficult to propagate
  - Root cause remains

- Switch to a secure by default implementation (flag for insecure/optimized)
  - No flag = secure implementation (potential performance loss)
  - All projects are patched at once
Mitigations

Two choices:

• Patch this particular issue by adding the proper flag
  • Most projects use the bignum API, not the whole SRP
  • **Difficult to propagate**
  • Root cause remains

• Switch to a secure by default implementation (flag for insecure/optimized)
  • No flag = secure implementation (potential performance loss)
  • All projects are patched at once
• Practical attack against SRP implementations
  • Vulnerability inherited by lots of projects
  • Easy to exploit because we can use each recover bits independently

Long term lesson: be careful with SCA, especially in PAKE implementation
• Practical attack against SRP implementations
  • Vulnerability inherited by lots of projects
  • Easy to exploit because we can use each recover bits independently

Long term lesson: be careful with SCA, especially in PAKE implementation

• Leakage in a weak generic function
  • Other protocols with small base may also use it
  • Contact use if you think of one!
Thank you for your attention!

https://gitlab.inria.fr/ddealmei/poc-openssl-srp

@daniel.de-almeida-braga@irisa.fr
1. Maps the victim’s address space

---

FLUSH+RELOAD\textsuperscript{1}

1. Maps the victim’s address space
2. Flush the instruction we monitor

1. Maps the victim’s address space
2. Flush the instruction we monitor
3. See how much time it takes to reload

---

1. Maps the victim’s address space
2. Flush the instruction we monitor
3. See how much time it takes to reload

---

1. Maps the victim’s address space
2. Flush the instruction we monitor
3. See how much time it takes to reload
   - Fast $\Rightarrow$ the victim already executed

---

1. Maps the victim’s address space
2. Flush the instruction we monitor
3. See how much time it takes to reload
   • Fast $\Rightarrow$ the victim already executed
   • Slow $\Rightarrow$ the victim did not

---