PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild

Daniel De Almeida Braga
Pierre-Alain Fouque
Mohamed Sabt
WAC’21 - August, 15\textsuperscript{th} 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 (WPA3 and CFRG competition after patents expiration) with practical security considerations

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

Small leakage can be devastating

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
\(^2\) D.Braga et al. *Dragonblood Is Still Leaking: Practical Cache-based Side-Channel in the Wild.* In ACSAC. 2020
\(^3\) J.Len et al. *Partitioning Oracle Attack.* In USENIX Security. 2021
What about SRP?

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

What about SRP implementations in the wild?
What about SRP?

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\textsuperscript{1}

\textsuperscript{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

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

Server

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

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

Commit

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

Verification

Verify client

Verify server

client_verify

server_verify
SRP Protocol Overview

Client:
- Choose a = rand(1, p-1)
- Compute A = g^a mod p
- Send user_id, A to Server

Server:
- Compute x = H(H(salt, H(id:pwd)))
- Compute v = g^x mod p
- Send salt, v = lookUp(user_id) to Client

Client:
- Compute B = kv + g^b mod p
- Send salt, B to Server
- Compute u = H(A||B)
- Compute x = H(salt||H(user_id:pwd))
- Compute S = (B-kg^a)^ux mod p
- Compute K = H(S)

Server:
- Compute u = H(A||B)
- Compute S = (Ag^u)^b mod p
- Compute K = H(S)

Verification:
- Client checks if S = (B-kg^a)^ux mod p
- Server checks if S = (Ag^u)^b mod p

Client verification:
- Verify server

Server verification:
- Verify client
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 modular exponentiation allowing to recover the password in a single measure.

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

No constant-time flag → Weak algorithm

Passive offline attack

No error and enough information


\(^2\) T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016
The Vulnerability
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 \times 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 \times 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
    ...
```
$bin(e) = 11010\ldots$

$res = g^e \mod p$

$w$ 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
...
```
**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)

```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

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

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

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

\[
\text{def} \ \text{BN\_mod\_exp\_mot\_word}(g, w, p):
\]

\[
\ldots
\]

\[
\text{w} = g \quad \# \ \text{uint64\_t}
\]

\[
\text{res} = \text{BN\_to\_mont\_word}(w) \quad \# \ \text{bigum}
\]

\[
\text{for} \ b \ \text{in range}(\text{bitlen}-2, 0, -1):
\]

\[
\rightarrow \text{next\_w} = w \times w
\]

\[
\quad \text{if} \ \text{next\_w}/w \neq w:
\]

\[
\quad \quad \text{res} = \text{BN\_mod\_mul}(\text{res}, w, p)
\]

\[
\quad \quad \text{next\_w} = 1
\]

\[
\quad \text{w} = \text{next\_w}
\]

\[
\text{res} = \text{BN\_sqr}(\text{res})
\]

\[
\quad \text{if} \ \text{BN\_is\_bit\_set}(x, b):
\]

\[
\rightarrow \text{next\_w} = w \times g
\]

\[
\quad \text{if} \ \text{next\_w}/g \neq w:
\]

\[
\quad \quad \text{res} = \text{BN\_mod\_mul}(\text{res}, w, p)
\]

\[
\quad \quad \text{next\_w} = g
\]

\[
\quad \text{w} = \text{next\_w}
\]

\[
\ldots
\]
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)

```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(w) - 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

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

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

\( w \) 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 \times 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 \times g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```
Optimized Square-and-Multiply

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

\( res = g^e \mod p \)

\( w \) 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

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

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

\( w \) 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

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

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

\( w \) 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 \times \ 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 \times g
        if next_w/g != w:
            res = BN_mod_mul(res, w, p)
            next_w = g
        w = next_w
    ...
```

Optimized Square-and-Multiply

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

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

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

Rules ($b \in \{0,1\}$):
- $Xxxx$ -> $111b$
- $Xxxxx$ -> $yyyyb$, $y \in \{110b, 10bb, 0111\}$
- $Xx....x$ -> $0 \ldots yyyyb$
Dictionary Attack

Client: \( x = H(\text{salt} \ || \ H(\text{user}_i\text{d:password})) \)

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

Recovered:

|     | 1 | 1 | 1 | b | y | y | y | y | b | 0 | y | y | y | b | 1 | 1 | 1 | b | 0 | y | y | y | b |
| pwd_1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| pwd_2 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 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 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| pwd_4 | 1 | 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 |
| 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 |
| ...   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| pwd_n | 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 |

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

- \(X_{xxx} \rightarrow 111b\)
- \(X_{xxxx} \rightarrow yyyyb, y \in \{110b, 10bb, 0111\}\)
- \(X_{x...x} \rightarrow 0 ... 0yyyyb\)

<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:

|     | 1 | 1 | 1 | b | y | y | y | b | 0 | y | y | y | b | 1 | 1 | 1 | b | 0 | y | y | y | b |
| pwd_1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 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 | 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 | 1 |
| pwd_4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 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 |
| ...   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| pwd_n | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |

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

- \( Xxxx \rightarrow 111b \)
- \( Xxxxx \rightarrow yyyyb, y \in \{110b, 10bb, 0111\} \)
- \( Xx.....x \rightarrow 0 ... 0yyyyb \)

<table>
<thead>
<tr>
<th>Password</th>
<th>X value</th>
<th>Diff score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Classical Workflow

SRP Protocol

Victim
Classical Workflow

SRP Protocol

Victim

Spy process
Classical Workflow

SRP Protocol

Victim

Spy process

Trace parsing
Classical Workflow

- **SRP Protocol**
- **Victim**
- **Spy process**
- **Trace parsing**
- **Offline dictionary attack**
- **Leaked information**
Classical Workflow

SRP Protocol

Victim

Spy process

Leaked information

Offline dictionary attack

Trace parsing

Remaining passwords
Impact and Mitigations
Impacted Projects

• Lots of projects using OpenSSL are impacted, including
  • OpenSSL TLS-SRP
  • Apple HomeKit ADK
  • PySRP (e.g. ProtonMail python client)
  • GoToAssit (?)

• Many reference libraries are based on OpenSSL to manage bignums
  • Ruby/openssl
  • Javascript node-bignum
  • Erlang OTP
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
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

OpenSSL’s choice
Conclusion
Conclusion

• 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
Conclusion

- 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

- This modular exponentiation is not limited to SRP
  - 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