Practical Cryptanalysis of Json Web Token and Galois Counter Mode's Implementations

Quan Nguyen (quannguyen@google.com)

Google Information Security Engineer (ISE)

My Job

- Conduct security reviews, i.e., play the ***attacker*** role mentioned in academic papers.

Agenda

- Json Web Signature/Encryption (go-jose) Security Review
  - How tricky and complicated RFC *design* leads to an unsafe implementation
- Galois Counter Mode (GCM) Crypto Bugs in OpenSSL GCM's wrapper, OpenJDK8, BouncyCastle, Conscrypt
  - We don't talk about well-known IV reuse issue but 2 *other* types of bugs that leak authentication keys.
  - GCM is fragile but its implementations were rarely checked.

Responsible Disclosure

- Square Inc. awarded me $5500 for go-jose's crypto issues.
- GCM bugs were reported to upstream developers and were acknowledged in Nexus Security Bulletin [1], Oracle Critical Patch Update [2], [3]

Important Observations

- Encryption/Signature signing’ input is mostly under our control
- *Decryption/Signature verification’ input is always under attacker’s control*

Json Web Signature/Encryption

- Json tokens that provides (multiple) signatures, ECDH, CBC-HMAC encryption

header	.	payload	.	signature

- Square Inc's go-jose is widely used by Google, Let's Encrypt, Square Inc, etc

Embedded public key in signature

- RFC7515, section 4.1.3: “The ‘JWK’ (JSON Web Key) Header Parameter is the public key that corresponds to the key used to digitally sign the JWS.”
- Attacker can generate private/public key pair and send the public key together with the signature and make the signature valid. See [jose] High risk vulnerability in RFC 7515
- Design level’s mistake by RFC.

Square’s go-jose embedded public key in signature

- Go-jose’s signing:
  - Enable embedded ‘JWK’ ***by default***
- Go-jose’s verification:
  - Exposes API to get ‘JWK’ out of signature and uses it for verification.
  - Does not even check whether ‘JWK’ is a public key; it accepts ***HMAC key!***
  - Has multiple sample tests to use embedded public key to verify.
- Not strictly a library’s vulnerability but easily misused

Go-jose's ECDH

- ❖ Checks well-known "Invalid Curve Attack" [1]
- ❖ To prevent attack: for NIST curves, check whether public key is on the *private key*'s curve.
- ❖ Go-jose, ECDH_ES (ephemeral static ECDH):
  - ➢ *Vulnerable*
  - ➢ *Sender can extract receiver's private key*

[1] Ingrid Biehl, Bernd Meyer, Volker Müller ,"Differential Fault Attacks on Elliptic Curve Cryptosystems", CRYPTO 2000

Go-jose's CBC-HMAC

HMAC	aad	16-byte nonce	ciphertext	*uint64(len(aad) * 8)*

- Found a few integer overflows in 32-bit machine, e.g.:

```
make([]byte, len(aad)+len(nonce)+len(ciphertext)+8)
binary.BigEndian.PutUint64(buffer[n:], uint64(len(aad)*8))
```

- Note: the correct instruction is uint64(len(aad))*8. uint64(len(aad))*8 makes the **boundary** between aad and nonce ***unambiguous***.
- Don't know how to turn integer overflows to remote code execution in go-lang
- How to turn integer overflows to HMAC bypass?

Go-jose's HMAC Auth Bypass Exploitation

$$\mathrm{HMAC}(aad \,||\, nonce \,||\, ciphertext \,||\, uint64(len(aad) * 8))$$

Buffer	aad	nonce	ciphertext	64

||

Buffer	newAad	newNonce	newCiphe rtext	64

- Denote: buffer = aad || nonce || ciphertext || 64,
- Assume attacker observes on the wire *aad, nonce, ciphertext* with
  - len(aad) = 8 (hence *uint64(len(aad)*8)* = **64**)
  - len(nonce) = 16,
  - len(ciphertext) = 536870928 (doesn't matter, just large value)
- Attacker creates:
  - newAadSize := 536870920 (hence uint64(newAadSize*8) = **64** because of integer overflow)
  - newAad := buffer[:newAadSize]
  - newNonce := buffer[newAadSize : newAadSize+16]
  - newCiphertext := buffer[newAadSize+16:]
- The attacker can create a *new* set of aad, nonce, ciphertext (and hence plaintext) with *valid* HMAC *without* knowing the HMAC key.

Go-jose's Multiple Signatures Verify()

```
for _, signature := range obj.Signatures {
        ...
        err := verifier.verifyPayload(input, signature.Signature, alg)   (1)
        if err == nil {
                return obj.payload, nil
        }
}
```

(1): If one of the signatures is valid; Verify() method returns the payload

Go-jose's Multiple Signatures

- If one of the signatures is valid; Verify() method returns the payload
- What's wrong?
  - The signature not only covers the payload but also covers the integrity of *protected* header.

header	.	payload	.	signature

Exploitation

1. Attacker observes a protected header and payload with valid signature.
2. Attacker creates multiple signatures:
   a. The 1st one with invalid protected header (e.g. a new JWK public key) with invalid signature.
   b. The 2nd one has valid protected header and valid signature that he captured in step 1.

3. The victim calls Verify() method, the method returns **no** error because the 2nd signature is valid; the victim starts using the attacker-injected 1st protected header.

```
{“payload”:”...”, “signatures”:
[{“protected”:”jwk RSA key”, “payload”:”...”, “header”:{“kid”:”...”},
  “signature”:”Invalid signature”},
 {“protected”:”...”, “header”:{“kid”:”...”}, “signature”: “valid signature”}]}
```

Galois Counter Mode

- Authenticated Encryption With Associated Data (AEAD)
- GCM is fragile but its implementations were rarely checked.

Galois Counter Mode

Encryption Key: K

Authentication key: $H = E(K, 0^{128})$

Counter : Y_0 = IV -12 bytes || $0^{31}1$

Plaintext: P[0] 16- byte P[1] 16-byte

Ciphertext: $C[0] = P[0] \oplus E(K, (Y_0 + 1) \% 2^{32})$

$C[1] = P[1] \oplus E(K, (Y_0 + 2) \% 2^{32})$

Finite Field GF(2^{128}): polynomial modulo $1 + x + x^2 + x^7 + x^{128}$, operation *

Authentication tag : $(((C[0]*H \oplus C[1]) * H) \oplus length(P)) * H \oplus E(K, Y_0)$

$= C[0]*H^3 \oplus C[1]*H^2 \oplus length(P)*H \oplus E(K, Y_0)$

OpenSSL GCM's Wrapper

Safe code:

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, auth_tag.data());
```

Vulnerable code:

```
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, auth_tag.size(), auth_tag.data());
```

auth_tag is what you get on the wire; it's under ***attacker's control***.

Auth. Tag Truncation Attack: Attacker sends **1** byte auth_tag

GCM's Wrapped Around Counter

- Y_0 = IV -12 bytes || $0^{31}1$
- $C[0] = P[0] \oplus E(K, (Y_0 + 1) \% 2^{32})$
- $C[1] = P[1] \oplus E(K, (Y_0 + 2) \% 2^{32})$
- After 2^{32} blocks, the counter will be wrapped around causing counter collision → leaks plaintext and authentication key.
- This is different from usual IV-reuse issue because it happens *even if* users use *different* IVs.

OpenSSL, BouncyCastle, Conscrypt, OpenJDK8

- OpenSSL ✓
- Conscrypt ✓
- BouncyCastle x
- OpenJDK8 x
- BouncyCastle & OpenJDK8 *missed* the critical security check:
  - Especially dangerous in Java Cipher *streaming* API.

Classic Timing Vulnerability in OpenJDK8

```
for (int i = 0; i < tagLenBytes; i++)
        if (computedTag[i] != expectedTag[i])
            throw new AEADBadTagException("Tag mismatch!");
```

Authentication bypass *once* is *not* interesting; attacker wants *authentication key*

Classic Timing Vulnerability in OpenJDK8

- Authentication bypass *once* is *not* interesting; attacker wants *authentication key*
- Joux's "Forbidden IV" Attack [1]
  - *Encryption's* input is under our (users) control
  - *NOT* exploitable in practice, unless users shoot themselves in the foot
  - *NIST fixed it since 2007*
- Decryption's input is under *attackers control*
  - Exploitable in practice

[1] Antoine Joux. "Authentication Failures in NIST version of GCM". NIST Comment, 2006

Attacker **chooses collided IVs in** ***decryption***

- Sends 2 pairs with collided IV to *decryption oracle*:
  - (IV, C1)
  - (IV, C2)
  - length(C1) = length(C2) = 16
  - C1 ⊕ C2 = **1**
- **In particular: IV= 0^{16}, C1 = 0^{16}, C2 = $0^{15}1$**
- Use previous *timing-attack* to figure out the auth tags authTag1 of (IV, C1), authTag2 of (IV, C2)

Attacker chooses collided IVs in *decryption*

$authTag1 = E(K, Y0) \oplus (C1*H^2 \oplus length(C1)*H)$

$authTag2 = E(K, Y0) \oplus (C2*H^2 \oplus length(C2)*H)$ where H is authentication key

$authTag1 \oplus authTag2 = (C1 \oplus C2) * H^2 = \mathbf{1}.H^2 = H^2$

Finding a square root in $GF(2^{128})$ is enough to find H. Happy hacking!

Extra Bugs

GCM Short Tag Attack

- Short tag attack [1] → leaks authentication key
- Safe default should be 16-byte auth tag

[1] Niels Ferguson. “Authentication weaknesses in GCM”. NIST Comment, 2005

Check safe default

- Golang: 16-byte ✓
- BoringSSL: 16-byte ✓
- Conscrypt x
  - cipher.init(Cipher.ENCRYPT_MODE, new SecretKeySpec(key, AES), new IvParameterSpec(encryptCounter));
  - Uses ***12-byte*** auth tag
  - Cites RFC 5084. Whose fault?
- Search for "RFC 5084"; found a few more instances of it.

References

1. David A. McGrew and John Viega. “The Security and Performance of the Galois/Counter Mode (GCM) of Operation (Full Version)”. INDOCRYPT 2004
2. Niels Ferguson. “Authentication weaknesses in GCM”. NIST Comment, 2005
3. Antoine Joux. “Authentication Failures in NIST version of GCM”. NIST Comment, 2006.
4. Morris Dworkin. NIST Special Publication 800-38D. 2007
5. Ingrid Biehl, Bernd Meyer, Volker Müller ,“Differential Fault Attacks on Elliptic Curve Cryptosystems”, CRYPTO 2000
6. M. Jones, J. Bradley, N. Sakimura. RFC 7515. May 2015
7. M. Jones, J. Hildebrand. RFC 7516. May 2015
8. Square Inc’s go-jose. https://github.com/square/go-jose
9. Tim McLean. “Critical vulnerabilities in JSON Web Token libraries” . https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

Thanks for your attention!

Acknowledgements

It’s my honor to work with and *to learn from* cryptanalysts Thai Duong and Daniel Bleichenbacher.