Team: Password Is Password

Mohit Jangid, Eric Lewantowicz, Joseph Shaffer

CSE 5473

4/22/2018

**POODLE and Bleichenbarcher Padding Attacks**

**POODLE Attack**

Outcomes:

1. Learn the vulnerabilities of SSLv3 encryption and padding protocol and implement the protocol in a realistic manner with a Client and Server communicating over TCP stream sockets.
2. Learn the POODLE attack and implement it against our Client/Server SSLv3 implementation.
3. Consider possible defenses approaches and implement a defense against a POODLE attack.
4. Make a visually informative simulation that will help convey the vulnerabilities of SSLv3, the steps of the POODLE attack, and the steps of the proposed defense to the rest of the class.

- POODLE: Padding Oracle On Downgraded Legacy Encryption

- a variant of BEAST attack

- Attack takes advantage of vulnerability of SSL version 3.0 with CBC-mode

- Discovered Sep 2014 by Google Security team

- SSL 3.0 created 20 years ago, but still commonly supported by browsers

- If issues connecting with current TLS protocols, browsers would fallback to older protocol versions such as SSL3.

- one instigation method: MitM attacker causes connection failures, triggering client and server to fallback to SSL3

connection, then execute POODLEattack

- Initial mitigation:

-- introduce TLS\_FALLBACK\_SCSV to list of client supported cipher suites. This alerts a server to check if a client

controlled by a MitM attempts to negotiate a lower cipher suite than both client and server support, generating

an inappropriate\_fallback alert, a fatal error that will terminate the SSL connection.

- Solution: disable SSL 3.0 support altogether in all browsers

-- or disabling CBC-mode to support only RC4 stream cipher mode (Google also disabled RC4)

-- original Google Chrome fix was to only disallow fallback due to retry after connection failures in order not to

cause compatibility issues with valid legacy sites still using SSL3.

1. Attacker established as MitM

- Attacker must be able to modify network transmissions between the victim client and the server

- Attacker does not need symmetric session key

- Attacker cannot modify MAC, or server returns HMAC error; must get a padding error from the server

- GOAL: decrypt secret cookie

2. Attacker uses a “downgrade dance” between target victim and the victim’s destination web server

- First handshake offers highest protocol supported by client

- If first handshake fails, retry with earlier protocol versions

- Network glitches and active attackers can also trigger a downgrade

3. **SSLv3 CBC vulnerabilities**

- The block cipher padding is not deterministic and not covered by the Message Authentication Code

-- Uses MAC-Then-Encrypt (TLS does not, and changes padding scheme)

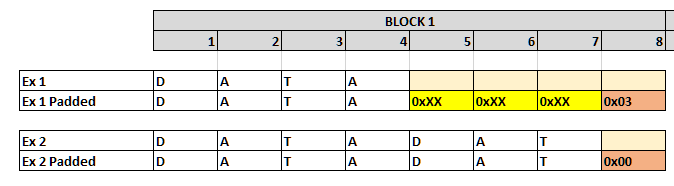
-- **problem: SSLv3 authenticates before padding and encryption, so a full padding block can be replaced!!**

- This means the validity of the padding cannot be fully verified during decryption

- Only one byte of padding is checked (e.g. the last byte of a full padding block) improves odds to 1/256 vs 1/2^128

- Padding 1 to B bytes (B is block size in bytes) is applied to create integral number of blocks before performing

CBC encryption on the blocks

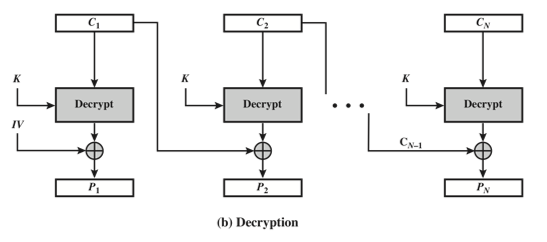


- Easiest to exploit if there’s an entire block of padding L­1 arbitrary bytes followed by a single byte of value L­1.



- IV||C0||C1||...||Cn are transmitted from client to server

- Server decrypts each block Pi = DK(Ci) ⊕ Ci-1 (symmetric session key, K, is the same per-connection)



- Server then checks for padding in Cn

- Server checks and removes MAC (recall MAC doesn’t include padding)

\* If there’s a full block of padding, and attacker replaces Cn (padding block) by any earlier ciphertext block Ci from

the same encrypted stream, the ciphertext will still be accepted if DK(Ci) ⊕ Cn­1 has 0x07 as its final byte

and rejected otherwise (SSL3 server becomes the Oracle)

- Use in HTTPS web setting to decrypt secure HTTP cookies using techniques from BEAST attack

- Attacker runs JavaScript agent on **http**://compromised-website.com that victim browses to

- Get the victim’s browser to send cookie-bearing HTTPS requests to **https**://website-to-get-access-to.com

- Attacker intercepts and modifies SSL records from victim browser in a way that the https site may accept

the modified record

- If the https site accepts the modified record, the attacker can decrypt one byte of the cookie

- Assume size of each block Ci is 16 bytes

- Assume that the size of the cookies is known

- MAC size in SSL3 CBC is 20 bytes

- This gives the format of the encrypted POST request:

ECBC[K, POST /**URI-path** **Cookie: name=value**...\r\n\r\n **body** ‖ **20­byte MAC** ‖ **padding**]

- Goal: decrypt Cookie bytes causing client to varying the cookie byte positions within subsequent requests

- Observe that MAC does not factor the padding, so the encrypted padding block can be replaced

- Attacker controls request /path and the request body, so attacker can cause the client to modify the request

before encryption and transmission such that:

1. Padding fills an entire block (encrypted into Cn)

2. Cookie’s first unknown byte appears as the final byte in an earlier block (encrypted into Ci)

- Attacker replaces Cn with Ci and forwards the modified SSL record to the server oracle

- When the server oracle rejects the record, the attacker tries again with a new request

- On average, server will accept 1 in 256 requests as valid when it recognizes valid padding byte

- Attacker concludes DK(Ci)[15] ⊕ Cn­1[15] = 15

Not explained in paper: (diagrams here)

DK(Ci)[15] ⊕ Cn­1[15] = 15 Attacker knows from Oracle

We can find Pi[15]:

D(Ci) = Pi ⊕ Ci-1 Known by CBC Encryption algorithm

D(Ci)[15] = Pi[15] ⊕ Ci-1[15]

Since DK(Ci)[15] ⊕ Cn­1[15] = 15,

( Pi[15] ⊕ Ci-1[15] ) ⊕ Cn­1[15] = 15 Substitute for DK(Ci)[15]

**Pi[15]** ⊕ Ci-1[15] ⊕ Cn­1[15] = 15 Solve for Pi[15]

- So **Pi[15]** = 15 ⊕ Cn­1[15] ⊕ Ci­1[15]

- This reveals the cookies’ first previously unknown byte.

- The attacker proceeds to the next byte by changing the sizes of request path and body simultaneously such that

the request size stays the same but the position of the headers is shifted, continuing until it has decrypted as

much of the cookies as desired

(some browsers send request header and request body in separate SSL records; for this case, only the path size

needs to be changed when proceeding to the next byte)

The expected overall effort is 256 SSL 3.0 requests per byte.

As the padding hides the exact size of the payload, the cookies’ size is not immediately

apparent, but inducing requests GET /, GET /A, GET /AA, ... allows the attacker to

observe at which point the block boundary gets crossed: after at most 16 such requests,

this will reveal the padding size, and thus the size of the cookies.

**POODLE Setup**

**Web Server**

execute web server from /Desktop/web\_server directory:

/usr/local/bin/python3.3 ssl3\_server.py:

open old firefox browser from /Desktop/old\_firefox/firefox/

firefox.exe

navigate to web server page:

**https://localhost:4443**

References

This POODLE Bites: Exploiting The SSL 3.0 Fallback, https://www.openssl.org/~bodo/ssl-poodle.pdf

[BEAST] T. Duong, J. Rizzo: “Here Come The ⊕ Ninjas”, 2011.