Agenda

1. TLS attacks timeline
2. Difficulty in reproducing attacks
3. Scapy-ssl_tls goals
4. Quick demo of scapy-ssl_tls capabilities
5. Custom TLS stacks, what to look for?
6. Fuzzing capabilities

Introduction

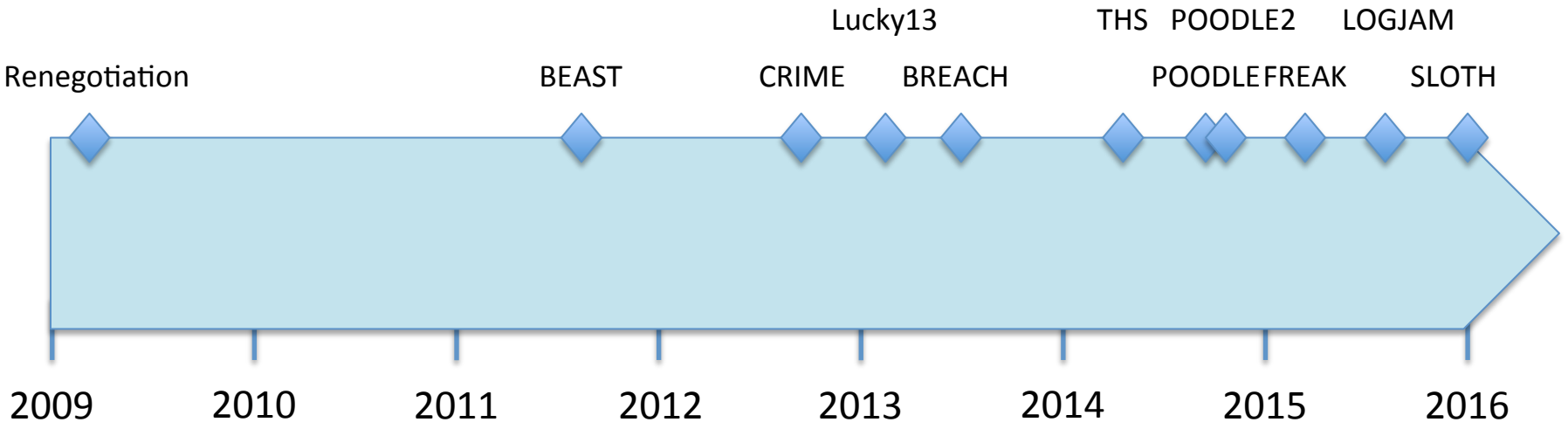
- TLS is a critical protocol to the internet
- Very few alternatives
- Session layer protocol for other protocols
- Very complex

TLS PROTOCOL LEVEL ATTACKS

Introduction

- Protocol under scrutiny
- Growth of the number of protocol level attacks
- Numerous implementation bugs

Timeline


Renegotiation
BEAST
CRIME
Lucky13
BREACH
THS
POODLE2
POODLE
FREAK
LOGJAM
SLOTH
2009
2010
2011
2012
2013
2014
2015
2016

Observations

- TLS protocol attacks increase:
  - Frequency
  - Complexity
- 2 classes:
  - Protocol level
  - Crypto level

REPRODUCING ATTACKS

Problems

- Understand the attack properly
- Practical impact (as opposed to theoretical problem)
- Reproducibility
- Fix (dev + Q&A)
- Fix for good (regression)

Response

- Customers do not always understand the practical impact
- Your response team has to provide a definite answer
- 2 solutions for custom implementations:
  - Crypto code review:
    - Lack of comparison point
    - Hard to get the full picture when deep into a crypto routine
  - PoC:
    - Lack of tooling
    - Big difference between regular lib and security focused lib

SCAPY-SSL_TLS

Introduction

- TLS & DTLS scriptable stack built above scapy
- Stateless (as much as possible)
- Packet crafting and dissecting
- Crypto session handling
- Sniffing (wire, pcap, …)

Why bother?

- TLS stacks are built to be robust
- Enforce input parameters to be valid
- Tear down connection on error
- Not very flexible

Goals

- Easy to install and use
- Simplify discovery and exploitation of TLS vulnerabilities
- Allow full control of any TLS field
- Tries very hard to maintain absolutely no state
- Good documentation and examples
- No checks or enforcements (up to user if desired)
- Sane defaults
- Transparent encryption

Concepts

- Start scapy
- All classes start with TLS:
  - Allows easy autocomplete
- What fields are available in a given TLS record?
  - ls(TLSClientHello)
- TLSSocket() is used to wrap the TCP socket
  - This is your base element to send/recv traffic
- Build packets scapy style:
  - p = TLSRecord()/TLSHandshake()/TLSClientHello()

Packet crafting/parsing

DEMO

A simple TLS 1.3 client

```
with TLSSocket(client=True) as tls_socket:
    try:
        tls_socket.connect(ip)
        print("Connected to server: %s" % (ip,))
    except socket.timeout:
        print("Failed to open connection to server: %s" % (ip,), file=sys.stderr)
    else:
        try:
            server_hello, server_kex = tls_socket.do_handshake(tls_version, ciphers, extensions)
            server_hello.show()
        except TLSProtocolError as tpe:
            print("Got TLS error: %s" % tpe, file=sys.stderr)
            tpe.response.show()
        else:
            resp = tls_socket.do_round_trip(TLSPlaintext(data="GET / HTTP/1.1\r\nHOST: localhost\r\n\r
\n"))
            print("Got response from server")
            resp.show()
        finally:
            print(tls_socket.tls_ctx)
```

Viewing a packet

```
###[ TLS Record ]###
  content_type= handshake
  version= TLS_1_0
  length= 0x36
###[ TLS Handshakes ]###
     \handshakes\
      |###[ TLS Handshake ]###
      |  type= client_hello
      |  length= 0x32
      |###[ TLS Client Hello ]###
      |     version= TLS_1_2
      |     gmt_unix_time= 1494985553
      |     random_bytes= '\xa6;\x10{\x0f\x7fU\x88\xeaH\xc6\xaf\xf8\xe4\xed\xd56\x07\xcf\xd6\x85\xc1^\xbd\x1f\xa3\x02\x85'
      |     session_id_length= 0x0
      |     session_id= ''
      |     cipher_suites_length= 0x2
      |     cipher_suites= ['ECDHE_RSA_WITH_AES_256_GCM_SHA384']
      |     compression_methods_length= 0x1
      |     compression_methods= ['NULL']
      |     extensions_length= 0x7
      |     \extensions\
      |      |###[ TLS Extension ]###
      |      |  type= supported_versions
      |      |  length= 0x3
      |      |###[ TLS Extension Supported Versions ]###
      |      |     length= 0x2
      |      |     versions= ['TLS_1_3_DRAFT_18']
```

A TLS 1.3 server

```
server_hello = TLSRecord() / TLSHandshakes(handshakes=[
    TLSHandshake() / TLSServerHello(version=version,
                                    cipher_suite=TLSCipherSuite.TLS_AES_256_GCM_SHA384,
                                    extensions=[key_share])])

client_socket.sendall(server_hello)
client_socket.sendall(TLSRecord() / TLSHandshakes(handshakes=[
    TLSHandshake() / TLSEncryptedExtensions(extensions=[named_groups]),
    TLSHandshake() / TLSCertificateList() / TLS13Certificate(
                                            certificates=certificates)]))

client_socket.sendall(TLSHandshakes(handshakes=[
    TLSHandshake() / TLSCertificateVerify(alg=TLSSignatureScheme.RSA_PKCS1_SHA256,
sig=client_socket.tls_ctx.compute_server_cert_verify())]))
```

WHAT TO LOOK FOR

Recon

- Fingerprint possible fork
- OpenSSL empty plaintext fragment
- JSSE, NSS stacked handshake
- Difference in Alert type when tampering with Finish message
- Alert is loosely defined, so stack specific

State machine

- Tricky testing: mostly manual work and knowledge of RFC
- Automated testing: FlexTLS:
  - Example: mono FlexApps.exe -s efin --connect localhost:8443
- Gives a good starting point for manual testing
- Lot of legacy stuff: server-gated cryptography anyone?

Diffie Hellman

- Check the validity of server (EC)DH params
  - Group size
  - Primality
  - Subgroup confinement attack (e.g: Off curve test (EC))
  - Signature algo used
  - …
- Send interesting values (small, non-prime, …)
- Scapy-ssl_tls uses TinyEC for EC calculation
- Allows to perform EC arithmetic

Side channels (RSA)

- Pre Master Secret is decrypted
- TLS mandates PKCS1 v1.5 for padding
- This needs to be constant time, see classic Bleichenbacher
- Time and Check for response difference on invalid padding (alert vs tcp reset)
- Can use pybleach pkcs1_test_client.py to generate faulty padding for your PMS

Side channels (ciphers)

- Padding and MAC checks must be constant time
- Alert type must be identical
- Time and check response when flipping bytes in padding and MAC
- Check for nonce reuse

Proper byte checking

- Some implementation only verify a few bytes of padding, MAC and verify_data (finish hash)
- All bytes must be checked for obvious reasons
- Send application data packets with flipped padding, MAC and verify_data
- Make sure you always get an alert

Fragmentation

- Any packet above 2**14 (16384) bytes must be fragmented
- But any fragment size can be chosen
- Some stacks don't cope well with TLS re-assembly
- Can be used to bypass devices which parse TLS, but fail-open
- Server can be requested to fragment using the Maximum Fragment Length Negotiation extension
- DTLS allows to specify the fragment offset in the handshake

TLS 1.3 specific

- Spec doesn't leave a lot of room for implementation errors
- Handling of 0-RTT, early data, PFS
- Resumption checks (SNI)

CONCLUSION

Strengths

- Scapy-ssl_tls can speed up PoC development
- PoC can be re-used as part of testing QA and regression
- Valuable to reproduce findings & develop mitigations
- Help in learning & experimenting with TLS

Thanks

- Thanks to tintinweb who started the project
- Bugs: https://github.com/tintinweb/scapy-ssl_tls/
- Contact:
  - Github: alexmgr

THANKS

IF TIME ALLOWS

Fuzzing

- Provides basic fuzzing through scapy
- Tries to be smart by preserving semantically necessary fields
- Use fuzz() function on any element

```
fuzz(TLSRecord()/TLSHandshake(type=TLSHandshakeType.SUPPLEMENTAL_DATA)/TLSAlert()).show2()
###[ TLS Record ]###
  content_type= handshake                 <= preserved
  version= 0x7391                         <= fuzzed
  length= 0x6                             <= preserved
###[ TLS Handshake ]###
     type= supplemental_data              <= overriden
     length= 0x2                          <= preserved
###[ Raw ]###
        load= '(\r'                       <= fuzzed
```

Fuzzing

- Only good for basic fuzzing
- Simple to plug in your own fuzzer
- Just generate data, scapy-ssl_tls takes care of the rest
- Good targets: TLS extensions, certificates, …

Examples

- The example section contains some useful base tools:
  - RSA session sniffer: given a cert, can decrypt wire traffic (like Wireshark)
  - Security scanner: a rudimentary TLS scanner (versions, ciphers, SCSV, …)
  - Downgrade test
  - …
- Just baselines to write your own tools