Cunning with CNG: 
Soliciting Secrets from Schannel
Why you might care

- Extracting TLS / SSL Keys (of various types) from memory
- Ability to decrypt TLS connections that use ephemeral key exchanges
  - For anything that uses Schannel: RDP, IE, Powershell, etc...pretty much anything .NET too
  - Past connections AND Future since the point of the cache is resumption
- Undocumented / partially documented structures elucidated
- TLS session caches mapped to the requesting processes, with SNIs
- A tool that does these things via Volatility/Rekall
- A paper that documents these things
How we get there

1. Briefest of TLS Refreshers
2. How Schannel Works
3. The Secrets :)
4. The Other Forensic Artifacts!
5. A live demo >.>
A Disclaimer

- This is NOT an exploit
  - It’s the spec! :D
- Microsoft has done nothing wrong
  - To the contrary, their documentation was actually pretty great
- Windows doesn’t track sessions for processes that load their own TLS libs
  - I’m looking at you Firefox and Chrome
- Windows doesn’t track sessions for process that don’t use TLS…
  - That’d be you teamviewer...
- This talk has nothing to do with Chanel
  - Sorry Aine.
The now infamous TLS Handshake
The now infamous TLS Handshake or, Session Resumption
Perfect Forward Secrecy  < and what it means to TLS

What we *want* to do

- One time use keys, no sending secrets!

What TLS *actually* does

- Caches values to enable session resumption
  - recommends `An upper limit of 24 hours is suggested for session ID lifetimes`
- When using the session ticket extension, sending the encrypted state over the network
  - basically returning to the issue with RSA, but using a more ephemeral key...

What implementations *also* do

- Store symmetric key schedules (so you can find the otherwise random keys...)
- Cache ephemeral keys and reuse for a while...

What’s an Schannel?

- It’s TLS -> the Secure Channel for Windows!
- A library that gets loaded into the “key isolation process” and the “client” process
  - Technically a Security Support Provider (SSP)
- Spoiler: the key iso proc is LSASS

- Microsoft’s CryptoAPI-Next Generation
- Introduced in Windows Vista
- Provides Common Criteria compliance
- Used to store secrets, also crypt them
  - The KSP & DPAPI for instance
- Important / reused keys are “isolated” from the less privileged/trusted “client” processes into the “key isolation process”
- Ncrypt is the “key storage router” and gateway to CNG Key Iso service

What the CNG?!
Schannel Cipher Suite Preferences

Schannel

by the docs

CNG Key Isolation

by the docs

Matching Session Keys

Basic Premis:

AES Keys are small and random

AES Key Schedules are larger and deterministic by design…they’re a schedule.

Most implementations calculate schedule once and store it*

While a connection is active, both side NEED access to the symmetric keys used for encryption/verification

*Based on this paper: http://citp.princeton.edu/pub/coldboot.pdf
Matching Session Keys

So I scanned LSASS for cross-matched AES key schedules on both hosts…

And got nothing.

Well, no matches anyway.
A friendly neighborhood P.S.A.

This announcement brought to you by an hour of wasted time
Matching Session Keys

**RDP MSTSC AES Keys [Client]**

**RDP SVCHost AES Keys [Server]**

The Session Key Structure

- Notice the value “3lss”
  - “3lss” -> “ssl3”
- Initially noticed while checking LSASS structs
- Structure is different in LSASS vs client process
- AES Key & Schedule highlighted
- Key and schedule appear multiple times in the same structure
The Session Key Structure

**_SSL_SESSION_KEY_**

- cbStructLength: 4
- dwMagic ["ssl3"]: 4
- dwProtocolVersion: 4/8
- pvCipherSuiteListEntry: 4/8
- IsWriteKey: 4
- pvBcryptKeyStruct: 4/8

**_BCRYPT_KEY_**

- cbStructLength: 4
- dwMagic ["UUUR"]: 4
- pvBcryptProvider: 4/8
- pvBcryptSymmKey: 4/8

**_MS_SYMMETRIC_KEY_**

- cbStructLength: 4
- dwMagic ["MSSK"]: 4
- dwKeyType: 4
- KeyLength: 4
- SymmetricKey: ?
- SymmKeySchedule: ?
The Ncrypt SSL Provider [ ncryptsslp.dll ]

Ncryptsslp Validation function Symbols

Master Key Validation Function Disassembly

These functions do three things:

- Check the first dword for a size value
- Check the second dword for a magic ID
- Return the passed handle* if all is good

*All handles in this case are explicitly pointers
<table>
<thead>
<tr>
<th>SSL Magic</th>
<th>Size (x86)</th>
<th>Size (x64)</th>
<th>Validation Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl11</td>
<td>0xE4</td>
<td>0x130</td>
<td>SslpValidateProvHandle</td>
</tr>
<tr>
<td>ssl12</td>
<td>0x24</td>
<td>0x30</td>
<td>SslpValidateHashHandle</td>
</tr>
<tr>
<td>ssl13</td>
<td>?</td>
<td>?</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>ssl14</td>
<td>0x18</td>
<td>0x20</td>
<td>SslpValidateKeyPairHandle</td>
</tr>
<tr>
<td>ssl15</td>
<td>0x48</td>
<td>0x50</td>
<td>SslpValidateMasterKeyHandle</td>
</tr>
<tr>
<td>ssl16</td>
<td>0x18</td>
<td>0x20</td>
<td>SslpValidateEphemeralHandle</td>
</tr>
<tr>
<td>ssl17</td>
<td>?</td>
<td>?</td>
<td>&lt;none&gt;</td>
</tr>
</tbody>
</table>
Ncryptssl “ssl3” symbols*

>_ Command - Dump C:\Defcon\Exa...  - [] X

0:000> .foreach(magic {s -[1]a
00007fff`df750000 00007fff`df76f000 3lss}){ln
magic}
nencryptssl!SlsGenerateSessionKeys+0x251
ncryptssl!SPSslDecryptPacket+0x43
ncryptssl!SPSslEncryptPacket+0x43
ncryptssl!SPSslImportKey+0x19a
ncryptssl!SPSslExportKey+0x76
ncryptssl!SPSslFreeObject+0x1b
ncryptssl!Ssl2GenerateSessionKeys+0x22c
ncryptssl!Ssl2GenerateSessionKeys+0x294

ssl3 = session key struct

Ncryptssl “ssl7” symbols*

>_ Command - Dump C:\Defcon\Exa...  - [] X

0:000> lmm schannel
start   end    module
name
00007fff`ed1e0000 00007fff`ed254000  schannel

0:000> .foreach(magic {s -[1]a
00007fff`df750000 00007fff`df76f000 7lss}){ln
magic}
nencryptssl!SPSslGenerateMasterKey+0x75
ncryptssl!SPSslGenerateMasterKey+0x5595
ncryptssl!SPSslGeneratePreMasterKey+0x15e
ncryptssl!TlsDecryptMasterKey+0x6b

ssl7 = pre-master secret struct?

*command output significantly reduced for brevity & clarity
## The Master Secret

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SSL_MASTER_SECRET</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbStructLength</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>dwMagic [&quot;ssl5&quot;]</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>dwProtocolVersion</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>dwUnknown1*</td>
<td>0/4</td>
<td>[alignment?]</td>
</tr>
<tr>
<td>pvCipherSuiteListEntry</td>
<td>4/8</td>
<td></td>
</tr>
<tr>
<td>bIsClientCache</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>rgbMasterSecret</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>dwUnknown2 [reserved?]</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

*Not present in x86 - either padding or part of previous member*
## The Master Secret

<table>
<thead>
<tr>
<th>Structure</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SSL_MASTER_SECRET</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbStructLength</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>dwMagic</td>
<td>4</td>
<td>[“ssl5”]</td>
</tr>
<tr>
<td>dwProtocolVersion</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>dwUnknown1*</td>
<td>0/4</td>
<td>[alignment?]</td>
</tr>
<tr>
<td>pvCipherSuiteListEntry</td>
<td>4/8</td>
<td></td>
</tr>
<tr>
<td>bIsClientCache</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>rgbMasterSecret</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>dwUnknown2</td>
<td>4</td>
<td>[reserved?]</td>
</tr>
</tbody>
</table>

*Not present in x86 - either padding or part of previous member*
Master Secret Mapped to Unique Identifier

- The Master Key is linked back to a unique ID through an "NcryptSslKey"
- The NcryptSslKey is referenced by an "SessionCacheItem"
- The SessionCacheItem contains either the SessionID, or a pointer and length value for a SessionTicket

At this point, we can identify and decrypt sessions robustly.
Schannel $  

Under the covers

'\_SSL\_SESSION\_CACHE\_CLIENT\_ITEM': [ 0x140, {
    'Vtable': [0x0, ['pointer64', ['void']]],
    'NcryptKey': [0x10, ['pointer64', ['void']]],
    'PublicCertificate': [0x18, ['pointer64', ['void']]],
    'PublicKey': [0x28, ['pointer64', ['void']]],
    'NcryptSslProv': [0x60, ['pointer64', ['void']]],
    'SessionIdLen': [0x86, ['short short']],
    'SessionId': [0x88, ['array', 0x20, ['unsigned char']]],
    'ProcessId': [0xa8, ['unsigned long']],
    'MaxLifeTime': [0xb0, ['unsigned long']],
    'CertSerializedCertificateChain': [0xb0, ['pointer64', ['void']]],
    'UnkList1Flink': [0xb8, ['pointer64', ['void']]],
    'UnkList1Blink': [0xc0, ['pointer64', ['void']]],
    'UnkCacheList2Flink': [0xc8, ['pointer64', ['void']]],
    'UnkCacheList2Blink': [0xd0, ['pointer64', ['void']]],
    'ServerName': [0xf8, ['pointer64', ['void']]],
    'CSessCacheManager': [0x110, ['pointer64', ['void']]],
    'SessionTicket': [0x128, ['pointer64', ['void']]],
    'SessionTicketLen': [0x130, ['int']],
},]
Schannel $  

Under the covers
Schannel $ Under the covers
The Key Pairs

- The Server & Ephemeral Key Pairs use an identical structure
- The Key Type is compared with different values
  - ssl6 gets compared with a list stored in bcryptprimitives
  - ssl4 gets compared with a list stored in NCRYPTPROV
- The Key Storage Provider Key (KPSK) is referenced indirectly through an “Ncrypt Key” struct*

*Not to be confused with an NcryptSslKey struct
The Ephemeral Key Data

- Public Key is referenced by schannel! CEphemKeyData
- Private Key is not stored in natively usable format, but is accessible
- The KPSK structure pointed to references another structure with the magic “MSKY” that appears to be the EccKey structure
- The KPSK structure has details about the curve selection / other valuable info
The Server (RSA) Private Key

- KSPK structure pointed to by the server key mostly resembles the file from disk
- The public information is parsed into fields
- The DPAPI protected private key blob is loaded into memory
The Server (RSA) Private Key
CNG was introduced in Vista
The Vista cache is different
It’s kinda proto-CNG
Prior to Ncryptsslp (Sslp functions are in Ncrypt)
Instead of Classes, the cache is just a doubly-linked list
No RFC5088 support (no tickets)
The Forensic Context

- Active Connection = Security Context
- ProcessID for client process stored
- Server Name Indicator (SNI) stored in the cache as well
- Cache Lifetime of 10 hours
- Session IDs are arbitrary, but not always random
  - Schannel is the perfect example, can be fingerprinted
- If the system is a client, why would it have a server cache?
  - RDP for one, almost guaranteed to live 10 hours (unless there are 20,000 connections afterward)

Global Schannel Variables of Significance:

- `schannel!CSs1Globals::m_dwMaximumEntries`
- `schannel!CSs1Globals::m_dwClientLifespan`
- `schannel!CSs1Globals::m_dwServerLifespan`
- `schannel!CSs1Globals::m_dwSessionTicketLifespan`
Extracting the Secrets

- Volatility & Rekall plugins
- By default (no args):
  - will automatically find lsass
  - will scan the heap
    - Can be configured to scan Writeable VADs, or full VAS
  - dumps to stdout in wireshark format
    - Can dump verbose object as json
- Hoping to have functionality integrated into PowerShell module soon
  - Got busy :(<
Decrypting an RDP Session (Ephemeral 🕵️ XCHG)

Decrypting an RDP Session (Ephemeral 🔒 XCHG)
Fin

@TinRabbit
Questions?
Special Thanks

For general support, helpful comments, their time, and encouragement.

- **Áine Doyle** - Badass Extraordinaire (OCSC)
- **Dr. John-Ross Wallrabenstein** - Sypris Electronics
- **Dr. Marcus Rogers** - Purdue Cyber Forensics Laboratory
- **Michael Hale Ligh (MHL)** - Volexity
- **Tatiana Ringenberg** - Sypris Electronics