

# Stamp Out Hash Corruption, Crack All the Things!

Ryan Reynolds, Manager, Crowe Horwath, LLP  
Jonathan Claudius, SpiderLabs Security Researcher, Trustwave  
July 2012

## Abstract

This whitepaper is to serve as a supporting reference to the Black Hat USA 2012 talk, "Stamp Out Hash Corruption, Crack All the Things!". The focus of both the paper and presentation is to show how a number of Windows password extraction tools – Cain and Able, Metasploit, Creddump and many others – yield corrupt data when extracting password hashes from the Windows Registry. Both the paper and the presentation include the discovery process and a detailed description of the problem, as well as a solution for obtaining the correct hashes.

## Content Primer

The motivation behind obtaining password hashes from Windows-based systems is very similar to obtaining password hashes from any other operating system, service or application. Generally speaking, the focus of this process is either to transform a hash into the original clear-text version of the password or to be able to use that hash directly (perhaps via the pass-the-hash technique in Windows) to either validate the security of the password itself or to escalate privileges in the context of a malicious user.

When referring to Windows-based password hashes, there are two different hash types that this paper will focus on; LAN Manager (LM)-style hashes and NT LAN Manager (NTLM)-style hashes. LM hashing is the older of the two hashing algorithms and comes with a number of security flaws:

- Passwords are not case-sensitive
- Passwords have a maximum length of 14 characters
- Passwords are split into two 7-character portions, each of which is hashed separately, drastically reducing the number of potential hash keys
- Hashes are not individually salted

NTLM hashing, being the newer of the two algorithms, is stronger than LM hashing. It eliminates the first three shortcomings, but it is still not individually salted, leaving both algorithms susceptible to pre-computed dictionary attacks.

Two methods for extracting password hashes will be discussed: memory injection into the LSASS process space (“memory injection”) and reading of the SAM from the Windows Registry (“registry reading”).

LSASS injection is likely the most popular method for obtaining Windows password hashes, using tools such as pwdump6 and fgdump 2.1, and is generally accepted as the traditional method of obtaining hashes. However, LSASS injection does come with its share of shortcomings:

- Modern anti-virus (AV) controls commonly prevent this method
- Potential to cause a crash in the LSASS process

Registry reading is historically less popular but has recently been considered a preferred approach, despite having been around for quite some time (approximately 18 years), because it overcomes a number of issues presented by the memory injection method:

- It is typically not obstructed by AV, as registry access is allowed as part of normal activity on a Windows system
- It does not present the system stability concerns of loading foreign DLLs into the memory of critical system processes
- Hashes can be extracted from systems that are not running by copying the appropriate hive files

## Research Motivations

The motivation behind this research was to identify and eliminate the source of inconsistencies in Windows hashes retrieved during real-world penetration assessments.

During assessments, password hashes were often obtained by using the registry reading method. Occasionally, though, extracted hashes would appear corrupted – they did not work in pass-the-hash techniques and they could not be cracked, even when using rainbow tables. However, when reverting to using the memory injection method, as a sanity check, entirely different hashes would be received for the same accounts. LM and NTLM hashes from an example user are provided below, using both methods.

```
4500a2115ce8e23a99303f760ba6cc96 (BAD LM HASH)
5c0bd165cea577e98fa92308f996cf45 (BAD NTLM HASH)
```

Figure: 1A (via Registry Reading Method)

```
aad3b435b51404eeaad3b435b51404ee (LM HASH)
5f1bec25dd42d41183d0f450bf9b1d6b (NTLM HASH)
```

Figure: 1B (via Memory Injection Method)





010001009AC412C7DA10C788963DF9DF7E6B5EF401000100B0FD8B04845B3E6836EC62EDD3EC84CA0100010001000100 (hash data section)

Figure: 3A (LM and NTLM Hash Data Stored)

0100010001000100B0FD8B04845B3E6836EC62EDD3EC84CA0100010001000100 (hash data section)

Figure: 3B (Only NTLM Hash Data Stored)

The remaining data (hash data section) for both data structures change when LM hash data is not present. As seen in figures 3A and 3B, the hash data is simply stripped away and the start and end delineators ("01000100") are still present. This means that in order for tools to properly parse hash data in both scenarios, an extraction tool needs to make a decision about whether or not the LM hash data is present or not. Most registry extraction tools, in use today and included within scope of this research, use the following parsing logic:

1. If Hash Data Section > 40 bytes (0x28) then
  - lmmoffset = Hash Data Offset + 4 bytes (0x04)
  - ntmloffset = Hash Data Offset + 20 bytes (0x14)
  - Parse as if LM and NTLM hash data are present
2. Else If Hash Data Section > 20 bytes (0x14) then
  - ntmloffset = Hash Data Offset + 8 bytes (0x08)
  - Parse as if NTLM is present

*Note: The above 4 byte increments used in the offset calculations are used to skip the start and end delineators that are present in the data structure.*

When a tool employs the above logic to figures 3A and 3B, the end result is the LM and NTLM hash data elements from each structure:

9AC412C7DA10C788963DF9DF7E6B5EF4 (LM HASH DATA)  
B0FD8B04845B3E6836EC62EDD3EC84CA (NTLM HASH DATA)

Figure: 4A (LM and NTLM Hash Data Stored)

B0FD8B04845B3E6836EC62EDD3EC84CA (NTLM HASH DATA)

Figure: 4B (Only NTLM Hash Data Stored)

As seen above, when a tool employs this logic it can accurately extract password hash data for this user. After this information is obtained by a tool, the resulting hash data can then be passed to cryptographic algorithms to decode hash data and translate into typical LM and NTLM hashes. These hashes can then be supplied to a cracking tool like John the Ripper (JtR) to obtain the clear-text passwords.

As noted previously, the above parsing logic is used by nearly all the registry-based extraction tools examined. With that in mind, a closer look at the F and V data for



to be the hash data section. Below we provide the hash data section for the examples described in Figures 5A and 5B.

```
010001009AC412C7DA10C788963DF9DF7E6B5EF401000100B0FD8B04845B3E6836EC62EDD3EC84CA
0100010015F478C0D71D99AB56AB61F0921DE0EF9C21D096BE07202EDF579D32EF31DF178E47CFC1
80A85D50451DBBCD73DB89F3E81DC94989A51D23610F8669762EBFD5DF73B40F40B956835E95719E
0C18D4B27CAC2754CA807AD818CB4C27677A52621BA0A5AFB8CAA34AC3DFCDA8054B939514CD7E8A
51840220C7E1AF65C0865C015E517C522EAB6710181584F4E2D0652C01000100300772638DEB3458
51FF5B0CCA0123BB9B5C279A405AC24B0E98A583843488CD968264658858D5560A2047DB06FC1126
9C826D74B1EA6C1F2B6293F992E0360D562D62A1C091EDDC0C054E6A47881065C4F38C5CF8887812
46B88769BCE6E08E3ADBC06193EF250EC43775C8A5AE558A44F87484AED9BE0B73464DCDA257CC67
(hash data section)
```

Figure: 6A (LM and NTLM Hash Data Stored)

```
0100010001000100B0FD8B04845B3E6836EC62EDD3EC84CA0100010015F478C0D71D99AB56AB61F0
921DE0EF9C21D096BE07202EDF579D32EF31DF172549756090BA6CB58D6EB32C31E0714EB7CF5C2A
4073BEBF1C979A4CD4F07404747D0EAE50AB676696E6797F4E232C0F7CAC2754CA807AD818CB4C27
677A526201000100BB10DCCFE8681DD551FF5B0CCA0123BBB83FA6A3F659351C0E98A583843488CD
B5E1E55C3E5B22010A2047DB06FC11269C826D74B1EA6C1F2B6293F992E0360D562D62A1C091EDDC
0C054E6A47881065
(hash data section)
```

Figure: 6B (Only NTLM Hash Data Stored)

When we check the size of both hash data sections, for Figures 6A and 6B, they are both greater than 40 bytes (0x28) in length. What this means is that, regardless of whether LM hash data is present, we will always parse the hash data section as if LM and NTLM hash are present. If we follow this logic, then we end up parsing the following hash data from Figures 6A and 6B.

```
9AC412C7DA10C788963DF9DF7E6B5EF4 (LM HASH DATA)
B0FD8B04845B3E6836EC62EDD3EC84CA (NTLM HASH DATA)
```

Figure: 7A (LM and NTLM Hash Data Stored)

```
01000100b0fd8b04845b3e6836ec62ed (BAD LM HASH DATA)
0100010015f478c0d71d99ab56ab61f0 (BAD NTLM HASH DATA)
```

Figure: 7B (Only NTLM Hash Data Stored)

As seen in Figure 7A, this logic correctly parsed V data that contained LM and NTLM hashes, even with historical password hashes stored. However, Figure 7B shows that this logic does not correctly parse the data when it contains historical password hashes and only a NTLM hash.

It is that point that leads to the crux of the issue, the flawed assumption that a hash data length greater than 40 bytes indicates the presence of both LM and NTLM hashes. Under this flawed assumption, a tool tasked with parsing an NTLM hash only, followed by historical password hashes, will always incorrectly parse what it believes to be the first hash (actually only part of the hash), since it is using the incorrect offset. It will also then attempt to parse a second hash but get completely junk data because it is reading into another data structure (the historical hashes).





When examining these header values that describe whether or not a hash is present for either LM or NTLM, as seen in the above Figures 8A and 8B in blue, two values are present that when unpacked result in either a 0x04 or a 0x14. If 0x04, this means that a hash is not present and if 0x14, this means that a hash is present. Knowing this, a modified parsing algorithm was developed to work as follows:

1. Read 160 bytes (0xA0) from beginning of data structure
2. Then parse the next 4 bytes as an integer(lm\_header)
3. Read 172 bytes (0xAC) from beginning of data structure
4. Then parse the next 4 bytes as an integer(nt\_header)
5. Read 156 bytes (0x9c) from beginning of data structure
6. Then parse the next 4 bytes as an integer(X)
7. Hash Data Offset = X + 204 bytes (0xCC)
8. If lm\_header == 20 then
  - a. lm\_exists = true
  - b. lm\_offset = Hash Data Offset + 4
  - c. Parse LM
9. If nt\_header == 20 then
  - a. If lm\_exists
    - i. nt\_offset = Hash Data Offset + 24
    - ii. Parse NTLM
  - b. Else
    - i. nt\_offset = Hash Data Offset + 8
    - ii. Parse NTLM

Using the above logic, a tool will parse Figures 8A and 8B to obtain the correct hash data even with additional data present at the end of the data structure.

```

14000000 (lm_header)
14000000 (nt_header)
010001009AC412C7DA10C788963DF9DF7E6B5EF401000100B0FD8B04845B3E6836EC62ED
D3EC84CA0100010015F478C0D71D99AB56AB61F0921DE0EF9C21D096BE07202EDF579D32EF31DF17
8E47CFC180A85D50451DBBCD73DB89F3E81DC94989A51D23610F8669762EBFD5DF73B40F40B95683
5E95719E0C18D4B27CAC2754CA807AD818CB4C27677A52621BA0A5AFB8CAA34AC3DFCDA8054B9395
14CD7E8A51840220C7E1AF65C0865C015E517C522EAB6710181584F4E2D0652C0100010030077263
8DEB345851FF5B0CCA0123BE9B5C279A405AC24B0E98A583843488CD968264658858D5560A2047DB
06FC11269C826D74B1EA6C1F2B6293F992E0360D562D62A1C091EDDC0C054E6A47881065C4F38C5C
F888781246B88769BCE6E08E3ADBC06193EF250EC43775C8A5AE558A44F87484AED9BE0B73464DCD
A257CC67 (hash data section)

```

Figure: 9A (LM and NTLM Hash Data Stored)

```

04000000 (lm_header)
14000000 (nt_header)
01000100B0FD8B04845B3E6836EC62EDD3EC84CA0100010015F478C0D71D99AB
56AB61F0921DE0EF9C21D096BE07202EDF579D32EF31DF172549756090BA6CB58D6EB32C31E0714E
B7CF5C2A4073BEBF1C979A4CD4F07404747D0EAE50AB676696E6797F4E232C0F7CAC2754CA807AD8
18CB4C27677A526201000100BB10DCCFE8681DD551FF5B0CCA0123BBB83FA6A3F659351C0E98A583
843488CDB5E1E55C3E5B22010A2047DB06FC11269C826D74B1EA6C1F2B6293F992E0360D562D62A1
C091EDDC0C054E6A47881065 (hash data section)

```

Figure: 9B (Only NTLM Hash Data Stored)

As seen in figures 9A and 9B, the respective LM and NTLM hash header elements indicate (via 0x04 or 0x14) whether the hash exists or not, so a tool can now make the correct parsing decisions when it comes to reading through the hash data section. Once a tool applies the final steps (5-7 listed above), the correct hash data is obtained for both examples as follows:

```
9AC412C7DA10C788963DF9DF7E6B5EF4 (LM HASH DATA)  
B0FD8B04845B3E6836EC62EDD3EC84CA (NTLM HASH DATA)
```

Figure: 10A (LM and NTLM Hash Data Stored)

```
B0FD8B04845B3E6836EC62EDD3EC84CA (NTLM HASH DATA)
```

Figure: 10B (Only NTLM Hash Data Stored)

## Affected Tools and Origins

A large number of tools, which extract hashes from the registry were considered as part of this research. Tools that were confirmed as producing corrupted hashes when using the registry extraction method were as follows:

- Metasploit Hashdump Script
- Creddump
- Samdump2 1.0.1
- Cain and Able
- Pwdump
- Pwdump5
- Pwdump7
- FGDump 3.0
- l0phtcrack 6.0

Of these tools, there was a mix of both open-source and closed-source projects. By examining the source code from the open source tools, the hashes they produced and the hashes produced from the closed source tools, it was clear that similar logic was used by all of them, which resulted in the same incorrect hashes.

In tracing the origin of these tools, it was determined that Pwdump version 1 (Pwdump) was likely the first tool to reverse engineer the process of gathering hashes from the registry.

Being that Pwdump was an open-source tool, it was clearly a source of information and inspiration for other new tool authors that eventually began using this approach and associated logic for parsing registry data. By reading through tool change logs, blog posts and other online sources, the following relationship diagram was constructed to show how Pwdump had influenced these tools and how it's influence spread through generations of tools.

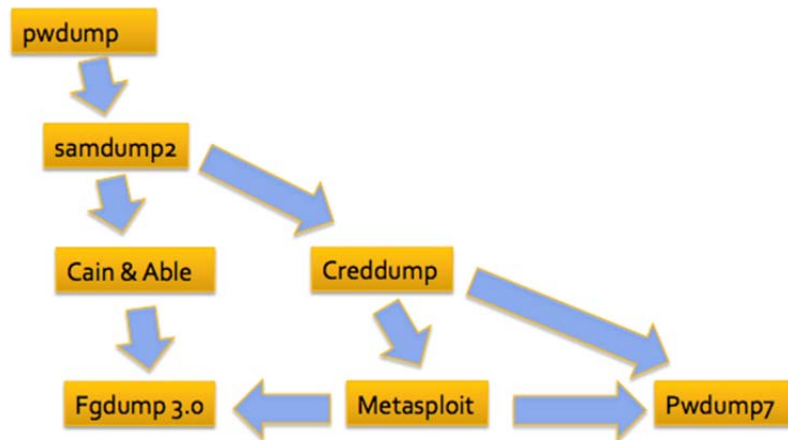


Figure: 11A

Although these relationships are important in showing how all these tools ended up using the similar logic, it is equally if not more important to understand the chronological time-line of when these tools were developed as seen in the following diagram.

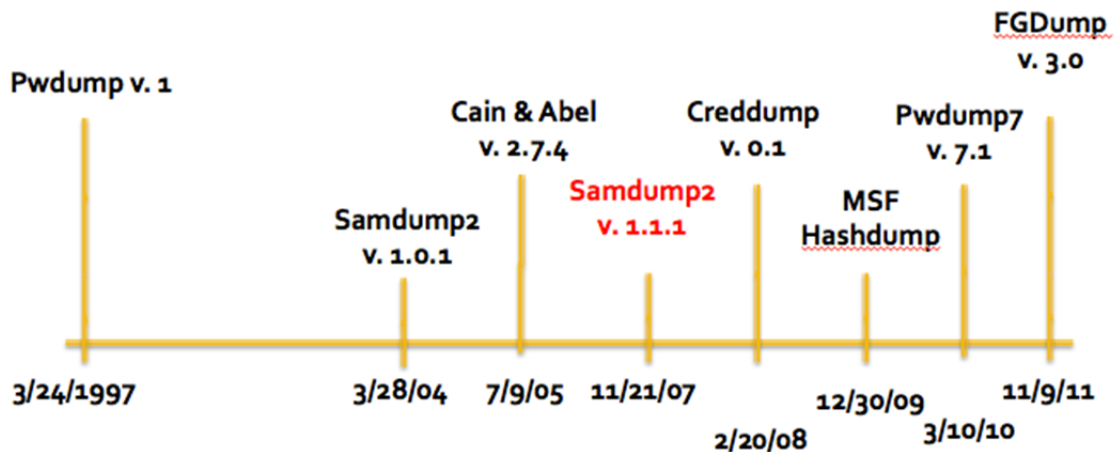


Figure: 11B

The above diagram contains two entries for samdump2 because in 2007, samdump2 identified that a flaw existed and developed a code fix for this issue in their 1.1.1 release. Ironically, 5 years later the tools that Samdump2 helped influence directly or indirectly, still (at the time of this writing) use the incomplete logic as implemented in the 1.0.1 release of Samdump2 and Pwdump version 1 as discussed in the technical section of this paper.

### Conclusions and Take Aways

Security professionals that are using extraction tools to obtain password hashes from Windows-based systems via the registry are regularly receiving corrupted

hashes. This is due to a logic flaw used by many tools that was described in detail within this whitepaper.

In addition to the identification of the flaw and its history, patches have been developed for both Metasploit and Creddump, which are both open-source. The goal here was to ensure that many of the tools described here are updated to utilize the improved logic described in this paper. To this end, an active outreach to closed-source tool developers in this space, such as Cain and Able, L0phtcrack, Pwdump7 and Fgdump, is already underway and some of these updates are already in development and should be available soon.

## Definition of Terms

**Hash** – The actual password hash (LM or NTLM) that is generated from Hash Data that represents the encrypted form of a clear-text password. This is what can be directly supplied to a cracking tool such as John the Ripper (JtR).

**Hash Data** – The source (or seed) data that is stored within the registry key “V” for each user that is transformed into either a LM or NTLM hash through a series of cryptographic algorithms. This data alone cannot be directly supplied to a cracking tool such as John the Ripper (JtR).

**Hash Data Section** – A subset of the V key stored in the SAM hive for each user that contains hash data, which has yet to be parsed into hash data elements.

## References

- Clark, Peter. "Security Accounts Manager." *Security Accounts Manager*. Beginningtoseethelight.org, 3 Apr. 2005. Web. 01 June 2012. <<http://www.beginningtoseethelight.org/ntsecurity/>>.
- Dolan-Gavitt, Brendan. "CredDump: Extract Credentials from Windows Registry Hives." *Push the Red Button*. Moyix.blogspot.com, 20 Feb. 2008. Web. 05 July 2012. <<http://moyix.blogspot.com/2008/02/creddump-extract-credentials-from.html>>.
- Dolan-Gavitt, Brendan. "SysKey and the SAM." *Push the Red Button*. Moyix.blogspot.com, 21 Feb. 2008. Web. 01 June 2012. <<http://moyix.blogspot.com/2008/02/syskey-and-sam.html>>.
- Dolan-Gavitt, Brendan. "Creddump-0.2.tar.bz2 - Creddump - Creddump 0.2 - Extracts Credentials from Windows Registry Hives." *Creddump Source Code Repository*. N.p., Feb. 2008. Web. 01 June 2012. <<http://code.google.com/p/creddump/downloads/detail?name=creddump-0.2.tar.bz2>>.
- Fizzgig. "Fgdump: A Tool For Mass Password Auditing of Windows Systems." *Http://fgdump.com/*. Fizzgig, 8 Oct. 2011. Web. 01 June 2012. <<http://fgdump.com/fgdump/>>.
- Moore, HD. "Bug #4402: Hashdump Script/post Module Breaks with Passwords Greater than 14 Characters." *Metasploit Framework Issue Tracker*. Rapid 7, 11 Mar. 2011. Web. 1 June 2012. <<http://dev.metasploit.com/redmine/issues/4402>>.
- Moore, HD. "Metasploit-framework / Scripts / Meterpreter / Hashdump.rb." *Metasploit Source Code Repository*. Rapid 7, 31 Dec. 2009. Web. 05 July 2012. <<https://github.com/rapid7/metasploit-framework/blob/4512089a34adafa05a477a5b86b911658d6b80ae/scripts/meterpreter/hashdump.rb>>.
- Moore, HD. "Safe, Reliable, Hash Dumping." *Metasploit Community Blog*. Rapid 7, 01 Jan. 2010. Web. 05 July 2012. <<https://community.rapid7.com/community/metasploit/blog/2010/01/01/safe-reliable-hash-dumping>>.
- Mueller, Lance. "Computer Forensics, Malware Analysis & Digital Investigations: Bypassing a Windows Login Password in Order to Boot in a Virtual Machine." *ForensickKB*. Forensickb.com, 22 Feb. 2008. Web. 1 June 2012.

<<http://www.forensickb.com/2008/02/bypassing-windows-login-password-in.html>>.

Oechslin, Philippe, and Cedric Tissieres. "Ophcrack - Samdump2 (samdump2-1.1.1.tar.gz)." *Ophcrack Source Code Repository*. SourceForge, 22 Nov. 2007. Web. 05 July 2012. <<http://sourceforge.net/projects/ophcrack/files/samdump2/1.1.1/>>.

"Oxid.it - Cain & Abel." *Oxid.it - Cain & Abel*. Oxid.it, n.d. Web. 01 June 2012. <<http://www.oxid.it/cain.html>>.

Rioux, Christien, Chris Wysopal, and Peiter Mudge Zatko. "L0phtCrack Password Auditor V6 - Documentation." *L0phtCrack Password Auditor*. L0phtcrack.com, n.d. Web. 01 June 2012. <<http://www.l0phtcrack.com/help/index.html>>.

"SAMInside." *InsidePro Password Recovery Software*. InsidePro Software, n.d. Web. 01 June 2012. <<http://www.insidepro.com/eng/saminside.shtml>>.

Tarasco, Andres, and Miguel Tarasco. "Password Dumper Pwdump7 ( V7.1 )." *Tarasco Security*. Tarasco.org, 03 Oct. 2010. Web. 05 July 2012. <[http://www.tarasco.org/security/pwdump\\_7/](http://www.tarasco.org/security/pwdump_7/)>.