
# Making Educated Guesses on Password Storage Using Timing Tests

Monika Morrow
Cindee Tran

# Introductions

- Monika Morrow
  - Senior Security Consultant at AppSec Consulting
  - https://github.com/monikamorrow
  - @FortyTwoWho
- Cindee Tran
  - Senior Security Consultant at AppSec Consulting

## Why?

- Detect insecure password storage without code review
- Allow external validation of your secure password storage

**Abstract:**
It is becoming more popular to limit password lengths to prevent denial of service by submissions of long passwords to computationally expensive password hashing algorithms like PBKDF2, bcrypt, and scrypt. How expensive is it really? Can we use denial of service tactics to learn about an applications password storage mechanism without a code review? Let's find out!

**The birth of an idea:**
The original problem: What is a reasonable maximum length password to recommend? This unearthed an interesting fact, Django limited its password length to 4096 to prevent Denial of Service attacks against its login which used PBKDF2. **Light bulb!** If a system using a secure password storage mechanism can be DoSed because of the computationally expensive algorithms it should be possible to determine if a system is using secure password storage functions by observing the amount of time the server takes to respond to password requests.

http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-much-length-can-be-bad-for-security/

# Passwords Then… (we wish!)

- Simple hash
  - MD5
  - SHA256
  - Downfall: Lookup tables
- Salted hash
  - Same algorithms as before
  - Solved: Lookup tables using salt
  - Downfall: Computing power overwhelming

## Passwords Now (we hope!)

- Work factor hashing algorithms
  - bcrypt
  - scrypt
  - PBKDF2
  - Solved: Computing power overwhelming using configurable work factor

bcrypt
maximum input length of 56 or 72 bytes
blowfish, expensive keysetup phase
scrypt
Relies on PBKDF2
PBKDF2
computationally expensive key derivation function

# Why a Work Factor?

- Hashing is FAST, measured in Mh/s
- SHA256 is slower than MD5 but PC3 still performs14,416,000,000 hashes per second

**Performance**

- PC1: Windows 7, 32 bit · Catalyst 14.9 · 1x AMD hd7970 · 1000mhz core clock · oclHashcat v1.35
- PC2: Windows 7, 64 bit · ForceWare 347.52 · 1x NVidia gtx580 · stock core clock · oclHashcat v1.35
- PC3: Ubuntu 14.04, 64 bit · ForceWare 346.29 · 8x NVidia Titan X · stock core clock · oclHashcat v1.36
- PC4: Ubuntu 14.04, 64 bit · Catalyst 14.9 · 8x AMD R9 290X · stock core clock · oclHashcat v1.35

| Hash Type | PC1 | PC2 | PC3 | PC4 |
|---|---|---|---|---|
| MD5 | 8581 Mh/s | 2753 Mh/s | 115840 Mh/s | 92672 Mh/s |
| SHA1 | 3037 Mh/s | 655 Mh/s | 37336 Mh/s | 31552 Mh/s |
| SHA256 | 1122 Mh/s | 355 Mh/s | 14416 Mh/s | 12288 Mh/s |
| SHA512 | 414 Mh/s | 104 Mh/s | 4976 Mh/s | 4552 Mh/s |

Credit: https://hashcat.net/oclhashcat/

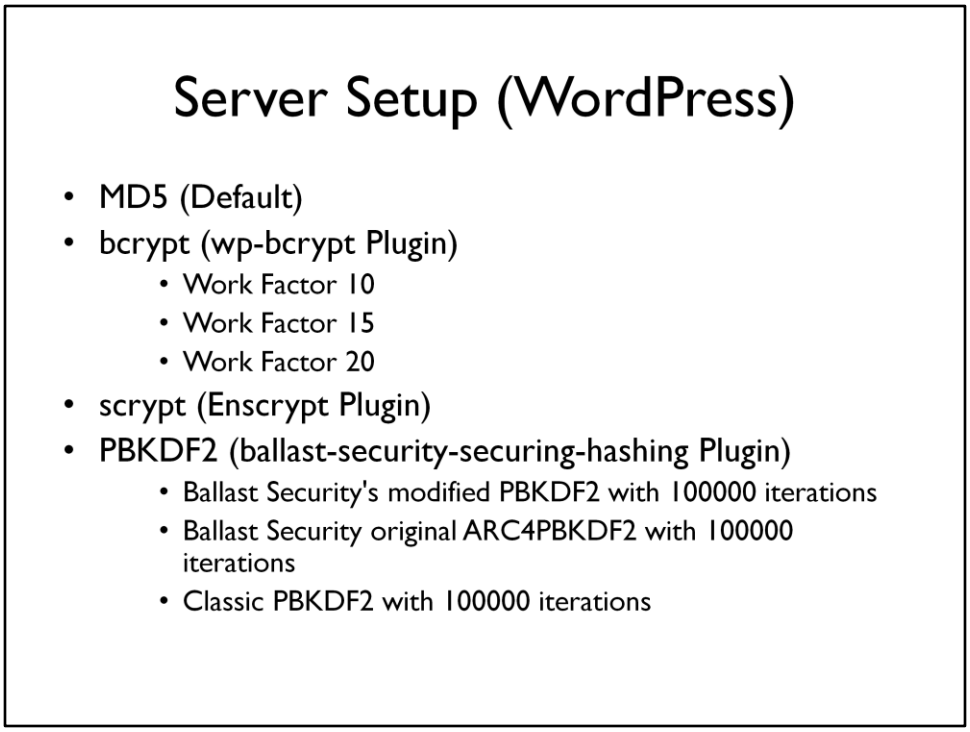
**Plaintext link:**
https://hashcat.net/oclhashcat/

Data appears to be circa early 2015

## Our Hypothesis

- The response delay introduced by the password hashing algorithm's work factor can be detected remotely.

**What we expected to learn:**
What approximate length password would reveal the likely underlying password storage mechanism?




To test our hypothesis, we set up an dedicated server hosting environment with 4 WordPress installations – one for each password hashing algorithm. For those that are unfamiliar with WordPress, it's just a simple open source CMS that allow you to easily install plugins so that you can extend or add functionality to the website.

By default for password storage, WordPress adds salt to the password and hashes it with 8 passes of MD5. For bcrypt, scrypt, and PBKDF2, we installed the appropriate plugins from their Plugin Directory and modified the code or settings to use different work factors. For bcrypt, we used three different work factors and for PBKDF2, we tested three different implementations of it.

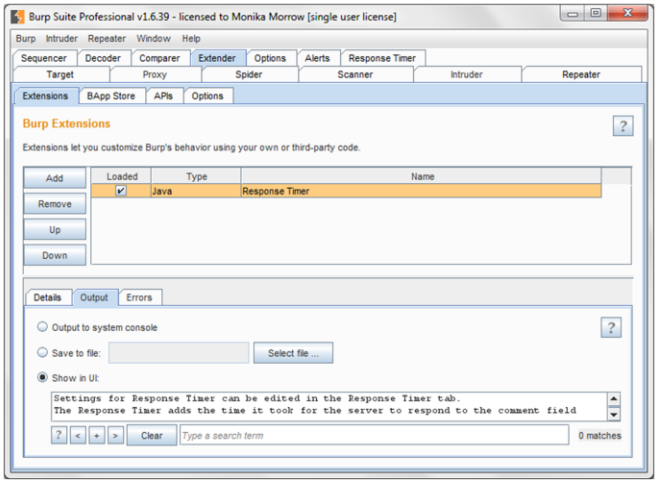
## Burp Extension

- Response Timer Extension
  - Average
  - Maximum
  - Minimum
- Data points
  - 100 requests per data point
  - 8 16 24 32 40 48 64 112 160
  - 208 256 4096 65536 1048576

In order to collect all the data for our research, Monika created an awesome Burp extension that captures and auto-populates the average, maximum, and minimum response times.

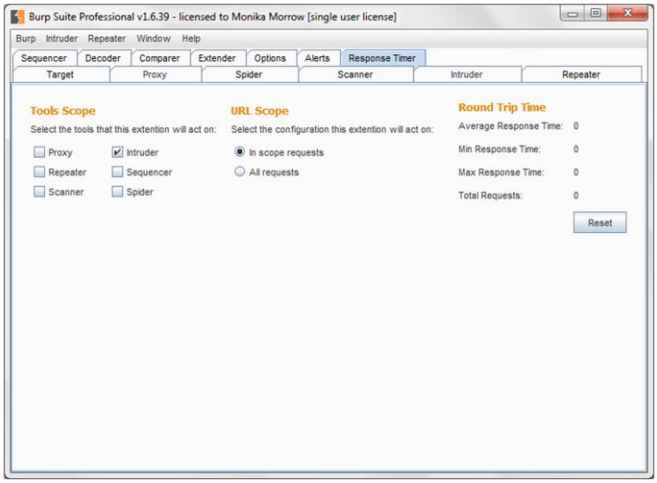
The data points represent the number of random characters we used for each password. For every data point, we tested for each of the 4 different password hashing algorithms and the different configurations for bcrypt and PBKDF2.

# The Setup – Load Extension



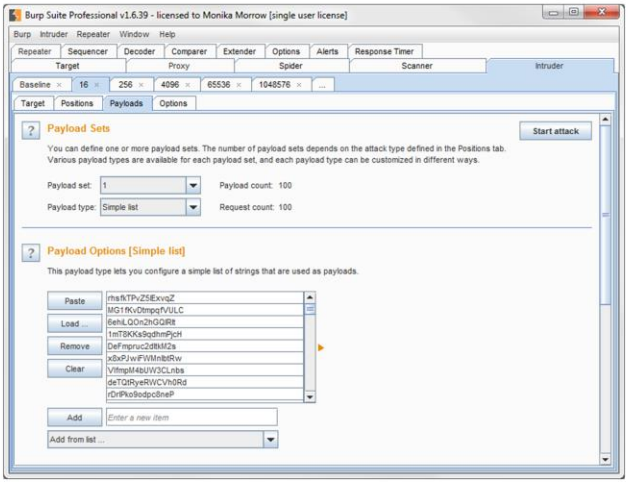

The Burp extension setup is simple, you just load the extension in Burp Extender.

# Configure Extension




To configure the burp extension, you need to keep Intruder checked under Tools Scope and also add the domain to scope under the Target tab or just select all requests.

# Configure Intruder - Payloads

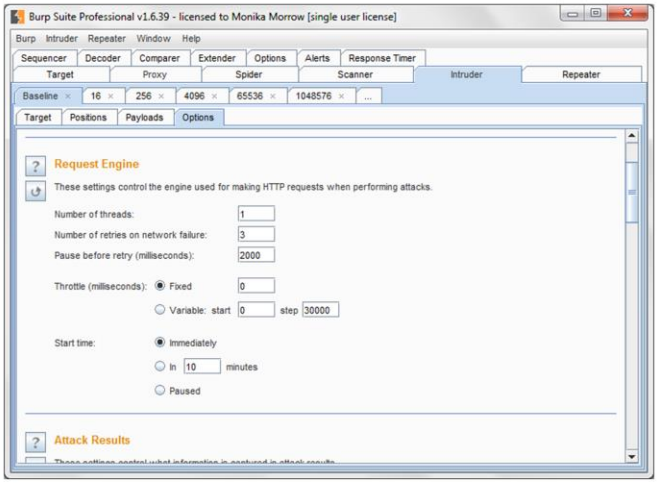



To configure Intruder, we generated random characters for each data point and configured Intruder to use a Simple List payload. The example here is for 16 characters.

**Example code for making random passwords**

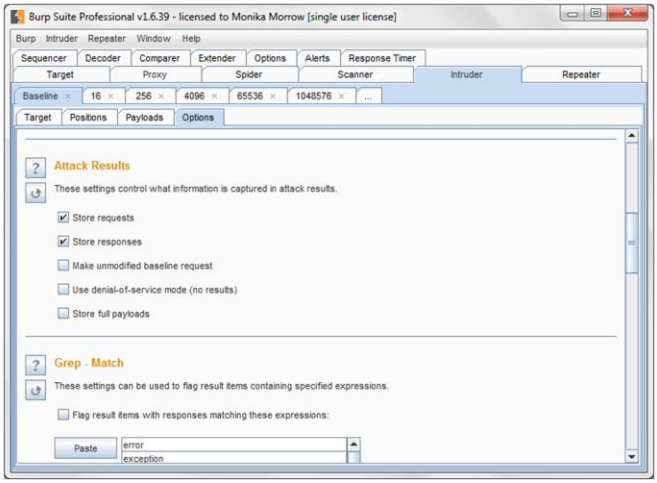
```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 20 ]; do
        cat /dev/urandom | tr -dc A-Za-z0-9 | head -c 1048576 >> 1m.txt
        echo \n >> 1m.txt
        let COUNTER=COUNTER+1
done
```

# Configure Intruder – Single Thread




In order to maximize our chances of getting valid results, we configured Intruder to run on single thread mode.

# Configure Intruder – No Base Request



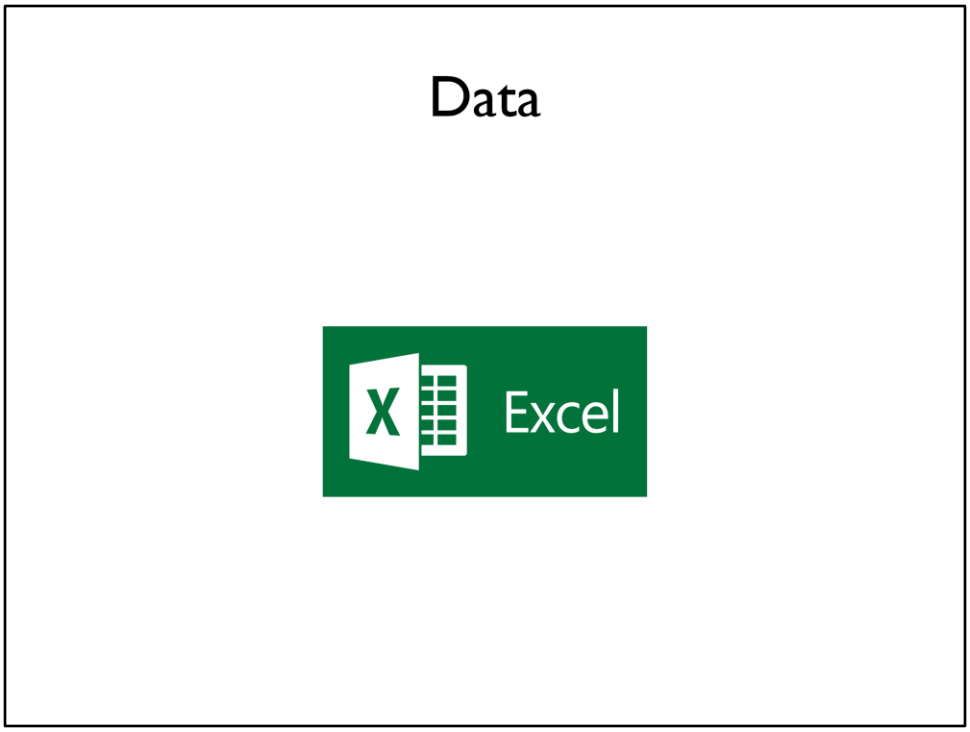

Another configuration we made was to ensure that the unmodified baseline request was unchecked to keep it from making a request with a shorter password and therefore adding invalid data to the data set.

## Data?

"I do know something,
just not with any certainty."

"I'm a scientist. Certainty is a big word.
I need time to collect my data."

-Dr. Amy Barnes
Volcano (1997)




**Plaintext link:**
https://github.com/MonikaMorrow/PasswordTimingTest/DataSet2016-03-26.xlsx

# What We Learned

- We have more to learn
- Large values don't have as much of an effect as we expected
- Results suggest measurable difference at small values

# What We Learned

- bcrypt only uses first 56 or 72 chars
  - Some applications hash passwords before performing bcrypt to ensure all characters are accounted for
- Ensuring limited maximum length password processed by PBKDF2 appears to limit DoS Risk

# Call For Moar Data!

- Reproduce our results
- Submit data
  - https://github.com/MonikaMorrow/PasswordTimingTest
- Contact us
  - MonikaMorrow@gmail.com
  - Tran.Cindee@gmail.com

**Plaintext link:**
https://github.com/MonikaMorrow/PasswordTimingTest

# Q&A