KEYS UNDER DOORMATS: PROBLEMS AND SOLUTIONS FOR SECURELY STORING CREDENTIALS IN WEB APPLICATIONS

DMITRIY BERYOZA

Who am I

Dmitriy Beryoza

dmitriy.beryoza@owasp.org

https://www.linkedin.com/in/beryozad

- Senior Security Researcher at *Vectra AI*
- Prior to that - Researcher/Pentester with X-Force Ethical Hacking Team at *IBM Security*
- 25+ years in software design and development
- Ph.D. in Computer Science, CEH, OSCP
- Interests: web and binary exploitation, reverse engineering, secure software development, network threat detection, and CTF competitions

Work presented here was done in collaboration with my former colleague - **Ron Craig, Program Manager in Secure Engineering, IBM Security**

Disclaimer

The views and opinions offered in this presentation do not necessarily reflect the official positions of either Vectra or IBM.

No responsibility is assumed by the author or his employer for your results or use of the information or code contained in or related to this presentation.

Agenda

- Introduction to secrets
- Incorrect ways to store secrets
- Abuse methods
- Proper secret storage
- Safeguarding the master key (KEK)
- Storing KEKs in the file system
- Introducing SideKEK
- Takeaways

Storage of Secrets is Common in Modern Apps

- Most applications today must handle a variety of secrets:
  - User login passwords
  - Back-end system credentials
  - Encryption keys
  - Private keys for certs
  - API keys
  - Sensitive customer data
- This data must be stored securely, encrypted *at rest* and *in transit*, with most sensitive data encrypted *in use,* too

Terms We'll Use in This Talk

A few definitions so we're all on the same page:

- ***Login password*** – A password or passphrase used by a **human** to access a front-end system. Store as a one-way randomly salted hash.
- ***Credential*** – A username and passphrase pair or other secret used by a **system** to access other resources. Must be encrypted.
- ***Data Encryption Key (DEK)*** – A cryptographic key used to encrypt secrets. These secrets may be anything, including cryptographic keys.
- ***Key Encryption Key (KEK) or "Master Key“*** – The top-level cryptographic key used to protect a Data Encryption Key

The guidance in this talk deals primarily with **credentials** rather than with ordinary end user passwords, or customer data.

Insecure Credential Storage is Common

- If you are developing software or doing code reviews, you have probably seen many incorrect ways of storing credentials
  - Embedded in source code

```
private static final String KEY="BeStKeYeVeR!!!11";
```

  - In plaintext in config files

```
[Database]
username=admin
password=You11N3v3rGue55
```

  - In plaintext in databases

```
mysql> select * from users
+----------+----------------+
| username | password       |
+----------+----------------+
| admin    | IAmInvincible! |
| guest    | guest          |
```

  - In plaintext in environment variables

```
setenv ENCRYPTION_KEY "47fHE9jikWq4daXo"
```

Obfuscation of Secrets Does Little for Security

- Secrets get Base64-encoded

```
String password="Q2FuJ3REZWNvZGVNZTotUA==";
```

- Secrets get XORed with constant key

```
for (int i = 0; i < secretPassword.length; i++)
    secretPassword[i] ^= 0xDB;
```

- Secrets get encrypted, but the encryption key is constant, hardcoded

```
IvParameterSpec iv = new IvParameterSpec("badinitvector:-(".getBytes("UTF-8"));
SecretKeySpec keySpec = new SecretKeySpec("worse key... :'(".getBytes("UTF-8"), "AES");
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(Cipher.DECRYPT_MODE, keySpec, iv);
byte[] original = cipher.doFinal(encryptedData);
```

Why is This a Problem?

- Plaintext sensitive data can be leaked from the application in a variety of ways (and yes, we consider encoded and obfuscated to be plaintext data)
- If data is encrypted, but the keys to decrypt the data are hardcoded or leaked, the data can still be accessed
- Keys that are hardcoded are hard to rotate periodically, or to change in the event of a breach

What Are Possible Ways for Abuse?

- Code analysis
  - Application source can be stolen
  - Application can be reverse engineered

- Leaking of files through vulnerabilities:
  - Path Traversal

    `https://myapp.com/get_log?name=../config/keystore.jks`

  - Local File Inclusion (LFI)

    `https://myapp.com/index?lang=../config/config.ini%00`

The Abuse Continues …

- XML External Entities (XXE)

```
<!DOCTYPE foo [
<!ELEMENT foo ANY>
<!ENTITY exploit SYSTEM

"file:///opt/myapp/keys/private.key"> ]>
<foo> &exploit; </foo>
```

- SQL Injection

```
https://myapp.com/profile?user='+and+1%3D2+union+all+
   select+load_file('/opt/myapp/data/customers.db')--
```

- Leaking of environment variables through vulnerabilities

```
https://myapp.com/get_log?name=../../proc/self/environ
```

- …and others

Keys Under Doormats

- Incorrectly stored encryption keys and credentials become “***keys under doormats***”:
  - Data not encrypted or encrypted poorly
  - When data is encrypted properly the key that protects it is often stored in close proximity to them
  - A vulnerability that allows file exfiltration may allow retrieval of both the encrypted data and the key that encrypts it
- In other words you either don't lock the door, or lock it but store the key nearby

What is the Proper Way to Store Secrets?

- No plaintext storage for data or keys
- No obfuscation, or custom-made encryption
- Keys must be generated fresh for every installed instance – otherwise one compromise can extend to entire customer base
- No hardcoding - one should have ability to change expired or compromised credentials/keys
- Use encrypted keystores/vaults
- Any password or key that is not needed in plaintext must be stored hashed, salted, and stretched to prevent brute force attacks

What is the Proper Way to Store Secrets? (cont.)

- Essentially, we want to store data in such a way that an attacker gaining an encrypted copy of it would not be able to read it

- A good guiding rule to follow is *Kerckhoffs's principle:*

  "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"

- Design your application in a way that full exfiltration of *code* and *encrypted data* **still** will **<u>not</u>** cause information compromise

Are We Back to Square One?

- There is still a problem, however…
- Credentials must be stored in a password protected keystore; sensitive data must be encrypted in the database, and so on
- A Data Encryption Key (DEK) is used to derive keystore passphrases, to encrypt database data, and to encrypt sensitive config values
- One additional key (sometimes called the Key Encryption Key, KEK, or Master Key) is used to strongly encrypt the Data Encryption Key
- *…but how do we safeguard the Master Key (KEK) itself?*
- *Déjà vu* and turtles all the way down
- If we store KEK in plaintext or hardcode it, we will have the very same problem as before. Theft of your data, keystore, DEK, *and the KEK* reveals everything

Can KEKs be Stored Safely?

- Unfortunately, many applications give up at this point and store the KEK in plaintext, hardcoded, or obfuscated

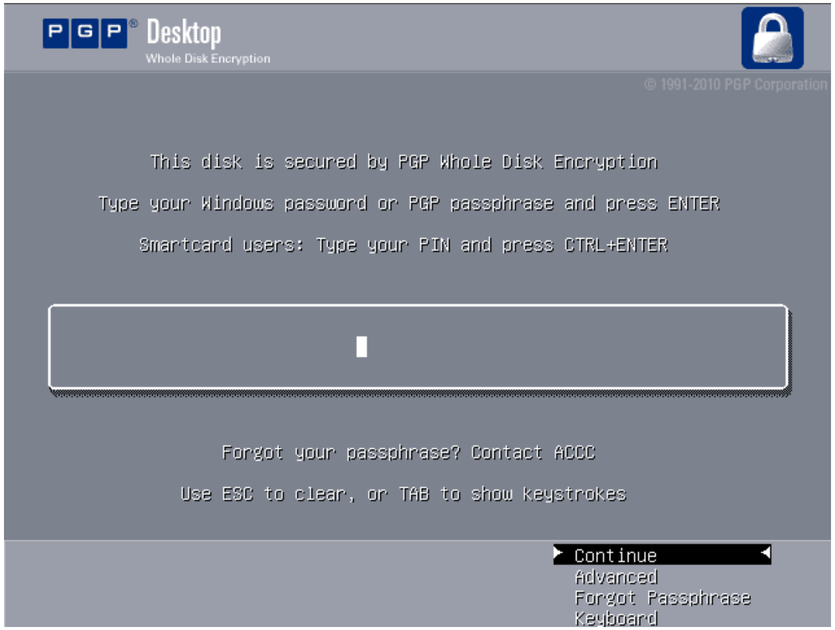
```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
InputStream keyStoreData = new FileInputStream("keystore.ks");
keyStore.load(keyStoreData, "s3cr3t_keyst0re_p4ssword!".toCharArray());
```

- This is not secure - KEKs must be stored in such a way that they cannot be stolen from a known file. Is that possible?

- Luckily, **YES**! There are multiple ways of doing that.

Best Practices for Storing KEKs

- **Not** storing the KEK (user-entered keys) at all
  - Some applications do not store the KEK at all and let the user enter the password KEK is derived from (e.g. full disk encryption apps)
  - Many applications must run in unattended mode and this method will <u>not</u> work for them



Best Practices for Storing KEKs (continued)

- Using **HSMs** / **Secure Enclaves**
  - *Hardware Security Modules* are specialized hardware devices that store encryption keys in hardware devices.
  - To interact with them one needs to run code in the context of the application, which means that to exploit them full application compromise must be achieved
  - HSM are not always available, and their cost of deployment may be too high for some applications

Best Practices for Storing KEKs (continued)

- Using **software HSM equivalents**
  - Software components similar in functionality to HSM
  - Usually part of OS but 3rd party implementations exist
  - Not built into all platforms
  - May have costs associated with deployment and management

- Using **Key Lifecycle Management** Systems
  - Specialized systems responsible for full cycle of key life, from creation to safe destruction
  - Have costs associated with deployment and management

Low Cost Ways of Storing KEKs

- Sometimes your application requirements do not allow the expense of dedicated HSM or other recommended ways of storing KEKs
- Are you then stuck with hardcoding the keys or writing them to a config file, and hoping for the best?

- **Not quite!** There are ways to store them based on unique characteristics of the system the application is running on
- Essentially KEK can be derived from system features that are hard for a remote attacker to determine

Low Cost Ways of Storing KEKs (continued)

- A number of unique IDs come to mind, but they **don't** make good candidates for a variety of reasons:
  - Network card MAC address
    - It is broadcast on the network and can be leaked; portions of it depend on the manufacturer; the possible key space is small
  - Motherboard ID, HDD ID, CPU ID
    - Key space may be small; some parts may be manufacturer-dependent; may be difficult to depend on in a virtual environment
  - Disk volume name
    - Key space may be small; may be leaked through *nix files

Low Cost Ways of Storing KEKs (continued)

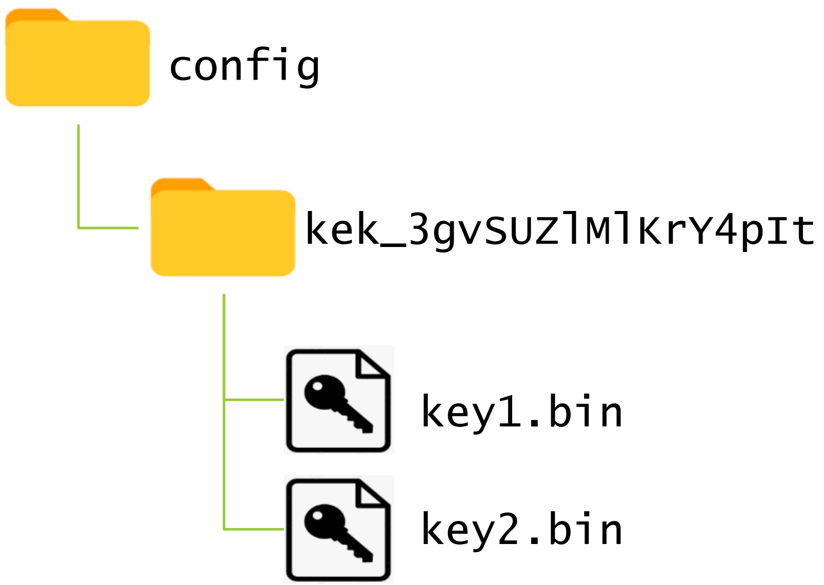
- Additionally, all these IDs were **not** generated in a cryptographically-secure way (too little entropy), and may not be under our control to be made secure

- It would be simplest to store a KEK *we* generate in the file system in some way
- *But* it cannot be stored in a file known to the attacker because there are too many ways to steal files

- In our research we found several methods to store KEKs by utilizing *features of the file system* that are not easy to discover by reading files
- We will outline two such methods that are easiest to implement

Storing KEKs in Secret Folders

- With this method of storage a ***subfolder with a random name*** is chosen to store KEK files
- The folder name is generated at install time as a long (filesystem-safe) random string
- Folder name also has a prefix by which it can be recognized (or suffix)
- KEKs are stored as regular files inside the folder
- *Application* can easily find the folder and load the keys



Storing KEKs in Secret Folders (continued)

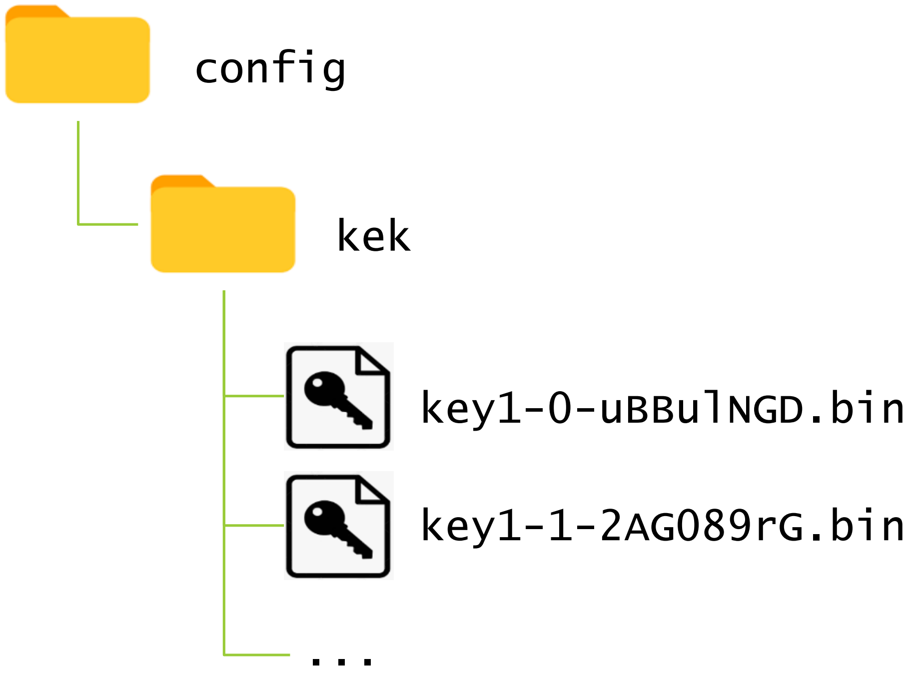
- Folder and file permissions should be set so that they are inaccessible to anyone except the user account the application runs under

- When the app needs to read/write the KEK, it searches for the folder with known prefix (or known extension) in the name

- *Attacker,* however, does not know the folder name and cannot use any of the file smuggling vulnerabilities to retrieve KEKs
- *Attacker* needs to have the ability to *search* folders, which most of the time would require code execution capabilities
- Such capabilities would likely mean full application compromise

Storing KEKs in File Timestamps

- A stealthier approach is to store key bytes in file ***timestamps***
- A key several bytes in size can be distributed over the timestamps of *multiple* files created for this purpose
- *Application* can interrogate timestamps of local files easily
- *Attacker* usually does not have access to file *metadata*, short of having code execution privileges on the system



Storing KEKs in File Timestamps (continued)

- Suppose we want to store an encryption key 32 bytes in size, and we choose to store 2 bytes per timestamp – we will need 16 files
- The files themselves can be empty, or garbage

- Multiple *sets* of files will have to be created to store multiple keys
- Key file names need to indicate *which* key the file stores. You can store the key alias in the extension (eg., *xxx*.key1, *yyy*.key2) or start the filename with the alias (eg., key1-*xxx*, key2-*yyy*), so you can search for the correct *set* of files for any particular key

- A way is needed to set the *order* the files are used to recover the key. Filenames can be sorted alphabetically to account for that.
- Example schemes:
- `key1-0-uBBulNGD78gTv53s.bin, key1-1-2AGO89jasi4fopnc.bin, ...`
- `F_l-yjbZJINfMugzUg…9gq.key1, Ujbf6HDTnc8ZHnUEs5…fuY.key1, ...`

Storing KEKs in File Timestamps (example)

- The following is a sample timestamp and its hex representation as stored in the file system:

```
08/30/2019 19:45:39.036    00 00 01 6C E4 0F D6 DC
```

- Suppose we wanted to store key 0xAABBCCDDEEFF
- We can create 3 files and embed bytes of the key in their timestamps (use *mtime* on *nix hosts):

```
08/19/2019 16:35:32.700    00 00 01 6C AA BB D6 DC
08/26/2019 07:39:46.268    00 00 01 6C CC DD D6 DC
09/01/2019 22:43:59.836    00 00 01 6C EE FF D6 DC
```

Storing KEKs in File Timestamps (caveats)

- Some of the systems (e.g. macOS) do not guarantee consistent setting of file modification timestamp milliseconds - better to use seconds granularity
- It is recommended to change only a portion of the timestamp so that it remains reasonably close to current time
  - Timestamps too far into the past or the future may confuse OS or applications
- Care should be taken in case of:
  - Backup/restore (may change the file timestamps)
  - Running utilities that modify timestamps (e.g. `touch`)
- KEKs may have to be backed up offsite for safekeeping
  - On a properly designed system destroying the KEK should **permanently deny access** to the data it protects

Additional Methods of Secret Storage

- Additional methods taking advantage of other file system features can be devised – feel free to experiment

- The goal is to utilize a method that is:
  - Resistant to a variety of vulnerability types (especially the ones that allow file exfiltration)
  - Ideally can only be defeated by attacker gaining code execution capabilities on the system
  - Low cost
  - Reliable in the presence of system changes

SideKEK

- Introducing ***SideKEK*** - a small open source Java library (and an OWASP project) to help you manage Key Encryption Keys in the file system:

  https://github.com/OWASP/SideKEK

- Implementations of the "secret folder" and "timestamp" techniques
- Plugs into Java security infrastructure – designed as Security *Provider* and *KeyStore* class implementations
- We welcome bug reports/feature suggestions/pull requests
- Feel free to contribute functionality in another language – the repository is structured to support it

Key Takeaways

- It is very easy to store secrets insecurely
- Special attention must be paid to proper encryption and key storage
- A wide variety of application vulnerability types can help achieve system compromise, particularly if they allow one to steal files
- “Master keys” (KEKs) suffer from the same problem of insecure storage as other secrets
- Currently recommended methods of KEKs storage are not free – they require additional hardware, software, or human resources
- By carefully using unique system characteristics (e.g. features of the file system) one can improve security of KEKs – and all other secrets by extension - inexpensively

Q&A

Acknowledgements

- **X-Force Ethical Hacking Team** members (Warren Moynihan, Chris Shepherd, Jonathan Fitz-Gerald, John Zuccato, Matt McCarty, Rodney Ryan, Troy E Fisher, Vincent Dragnea, Nathan Roane, Kamil Sarbinowski), *IBM Security*

- **Paul Ionescu**, Security Architect, Cloud and Data Center Security, *Trend Micro*

Thank you!