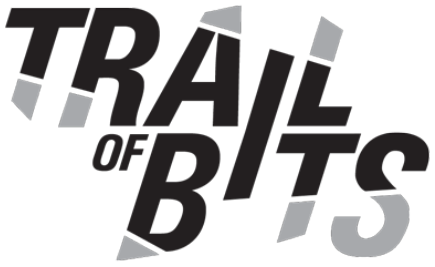

ETH2 Deposit CLI

Security Assessment

September 4, 2020

Prepared For:
Carl Beekhuizen | Ethereum Foundation
carl.beekhuizen@ethereum.org

Hsiao-Wei Wang | Ethereum Foundation
hww@ethereum.org

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Jim Miller | Trail of Bits
james.miller@trailofbits.com

Executive Summary

From August 24 through September 4, 2020, Ethereum Foundation engaged Trail of Bits to review the security of the ETH2 deposit tool. Trail of Bits conducted this assessment over the course of four person-weeks with three engineers working from commit `ec608ab` from the `ethereum/eth2.0-deposit-cli` repository.

Week one: We gained an understanding of the codebase, reviewed the code against the most common Python-related flaws, and looked at how the code performs key generation according to EIPs 2333, 2334, and 2335.

Final week: As we concluded our manual review, we focused on the correct use of cryptographic primitives and verification of best practices for compiling and releasing the deposit binary.

Our review resulted in 10 findings ranging from high to informational severity.
Of interest are some issues related to handling sensitive information, which may result in the loss of funds: They allow leakage of mnemonic phrases (TOB-EF-DEP-001), passwords (TOB-EF-DEP-006), and encrypted keys (TOB-EF-DEP-002). Additionally, two high-severity issues are related to the compilation (TOB-EF-DEP-005) and distribution (TOB-EF-DEP-001) of the deposit tool. Finally, we make several code quality recommendations in Appendix C.

Overall, the code follows a high-quality software development standard and best practices. It has suitable architecture and is properly documented. The interactions between components are well-defined. The functions are small, with a clear purpose.

We recommend addressing the findings presented and making sure that security best practices are followed during the binary releases, using signatures that can be independently verified by the final users.

Project Dashboard

Application Summary

Name	Deposit CLI
Version	`eth2.0-deposit-cli` commit: `ec608ab`
Type	Python
Platforms	Linux, macOS, Windows

Engagement Summary

Dates	August 24–September 4, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■ ■
Total Medium-Severity Issues	2	■ ■
Total Low-Severity Issues	5	■ ■ ■ ■ ■
Total Informational-Severity Issues	1	■
Total	10	

Category Breakdown

Access Controls	2	■ ■
Cryptography	1	■
Data Exposure	2	■ ■
Data Validation	4	■ ■ ■ ■
Undefined Behavior	1	■
Total	10	

Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning.

Category Name	Description
Function Composition	**Strong.** Functions and classes were organized and scoped appropriately.
Data Validation	**Moderate.** Although some data validation is performed, there are no checks for usage of weak passwords (TOB-EF-DEP-003). Additionally, we found instances in which the tool can crash, especially related to unexpected encoding of passwords (TOB-EF-DEP-008).
Cryptography	**Strong.** The use of cryptographic primitives is correct and follows security best practices.
Key Management	**Moderate.** We report a few concerns about passwords and accessing the secret keys protected by them (TOB-EF-DEP-002, TOB-EF-DEP-003, TOP-EF-DEP-006). Otherwise, we found no issues related to key generation.
Specification	**Strong.** Several EIPs describing the functionality of the tool were available. The codebase included a fair amount of comments explaining the purpose of each function.
Testing & Verification	**Moderate.** While the tests already integrated a number of static tools such as linters, the use of bandit revealed one high-severity issue (TOB-EF-DEP-005).

Engagement Goals

The engagement was scoped to provide a security assessment of the ETH2 deposit tool. Specifically, we sought to answer the following questions:

- Does the deposit tool comply with the `EIP 2333`, `EIP 2334`, `EIP 2335`, and `BIP39` standards?
- Can the private keys or password be leaked?
- Can the private keys be generated incorrectly?
- Can the mnemonic phrase be generated incorrectly?
- Can the `JSON` files be generated incorrectly?
- Are the cryptographic primitives used safely and correctly?
- Are any of the cryptographic primitives vulnerable to known cryptographic attacks?
- Does the tool install properly and behave identically across different operating systems?

Coverage

ETH2 deposit tool: The ETH2 deposit tool is used to generate and encrypt private keys for the validators. Trail of Bits reviewed deposit tool source code using manual review and static analysis tools. We focused our efforts on password handling, key generation, and storage. We looked for leaks of private information and insecure usage of cryptographic primitives. We also reviewed how the `EIP 2333`, `EIP 2334`, and `EIP 2335` are implemented in the tool to make sure they follow all the specifications.

Binary compilation and distribution: The ETH2 deposit tool is compiled using PyInstaller and released in GitHub. We reviewed how the deposit tool is compiled and released to make sure it follows security best practices.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

+ **Properly clear the terminal buffer after the tool is no longer used.** This will minimize the risk of users exposing sensitive information onscreen. The proper fix depends on the operating system and should be implemented carefully. In addition, either clear the clipboard after the tool has finished or block the user from copying the data altogether. TOB-EF-DEP-001

+ **Minimize the permissions when saving sensitive files.** This will mitigate the risk in case an attacker accesses the system used for key generation. TOB-EF-DEP-002

+ **Add a validation check on passwords to ensure minimum length.** This will mitigate the risk of an attacker brute-forcing the encrypted keys. TOB-EF-DEP-003

+ **Use strings instead of `JSON` numeric values to implement the amount field.** This will prevent any ambiguity when parsing the numeric fields of the deposit `JSON` files. TOB-EF-DEP-004

+ **Replace the use of assertion for proper input validation and do not include assertions with side effects.** This will enable the tool to run correctly even if the bytecode is optimized. TOB-EF-DEP-005

+ **Do not allow users to use the command line argument to specify passwords.** This will prevent the password from leaking to other users in the underlying operating system. TOB-EF-DEP-006

+ **Include signatures to allow users to verify the released binaries.** This will mitigate the risk of using malicious binaries if an Ethereum developer's Github account is compromised. We recommend using `gpg` with public keys uploaded to public servers. TOB-EF-DEP-007

+ **Validate the configuration of terminal encoding and system encoding** to mitigate the risk of users employing configurations that fail to run the tool properly. Additionally, document the recommended configuration for users. TOB-EF-DEP-008

+ **Make sure the keystore and deposit file have always matching filenames** so users won't be confused when selecting the deposit `JSON` file to upload. TOB-EF-DEP-009

+ **Determine which parameters are unsafe to ever use for `scrypt` and `PBKDF2`.** Once these are determined, prevent unsafe use by adding a warning that checks if either `scrypt` or `PBDKF2` are given unsafe parameters. TOB-EF-DEP-010

Long term

+ **Consider presenting the user with the secret mnemonic phrase for only a short period of time (e.g., a few minutes), and then clearing this information upon timing out.** This will minimize the risk of users exposing sensitive information onscreen. TOB-EF-DEP-001

+ **Review the amount of necessary permissions for each component of your system.** This will mitigate risk in case an attacker accesses the system used for key generation. TOB-EF-DEP-002

+ **Use a password strength library/entropy calculator on user passwords to enforce use of strong passwords.** This will mitigate the risk of an attacker brute-forcing the encrypted keys. TOB-EF-DEP-003

+ **Provide a recommended `JSON` implementation to use when interacting with the `JSON` files generated by the deposit tool.** This will prevent any ambiguity when parsing the numeric fields of the deposit `JSON` files. TOB-EF-DEP-004

+ **Review every feature of the Python language you use in case it demonstrates different behavior when the bytecode is optimized.** This will mitigate the risk of the tool running incorrectly if the bytecode is optimized TOB-EF-DEP-005

+ **Minimize the use of sensitive information and ensure secrets are not exposed via operating system logs** to avoid leaking the sensitive information to other users. TOB-EF-DEP-006

+ **Review the chain of trust of the precompiled releases used to generate private keys.** This will mitigate the risk of using malicious binaries if an Ethereum developer's GitHub account is compromised. TOB-EF-DEP-007

+ **Review the use of text encoding in each step of key generation and storage.** This will mitigate the risk of users employing configurations that fail to run the tool properly. Also, properly document each corner case, especially for non-Latin languages. TOB-EF-DEP-008

+ **Make sure users have enough information to know exactly which deposit file they want to upload** so users won't be confused when selecting the deposit JSON file to upload. TOB-EF-DEP-009

+ **Consider adding checks that prevent both `scrypt` and `PBKDF2` from being called with unsafe parameters.** Determining whether these parameters are unsafe depends on how these functions need to be used. TOB-EF-DEP-010

Findings Summary

#	Title	Type	Severity
1	Generated mnemonic could be leaked	Data Exposure	Low
2	Deposit stores a world-readable file with sensitive information	Data Exposure	Low
3	Deposit does not provide entropy validation on passwords	Access Controls	Low
4	Saving large `JSON` integers could result in interoperability issues	Undefined Behavior	Low
5	Use of `assert` will be removed when the bytecode is optimized	Data Validation	High
6	Passwords are accessible via shell history	Access Controls	Medium
7	PyInstaller binaries should be distributed with signatures	Data Validation	High
8	Certain encodings can make passwords impossible to input	Data Validation	Low
9	Naming of the resulting `JSON` files can be misleading	Data Validation	Medium
10	Python Crypto wrappings allow unsafe parameters	Cryptography	Informational

1. Generated mnemonic could be leaked

Severity: Low
Type: Data Exposure
Target: `eth2deposit/deposit.py`

Difficulty: High
Finding ID: TOB-EF-DEP-001

Description
The Deposit CLI tool is responsible for generating BLS key pairs. This tool implements a series of EIP standards that specify how to derive these key pairs from a mnemonic phrase. Specifically, a random mnemonic phrase is generated by the tool and then it is used to derive the BLS key pairs. This phrase is given to the user so their keys can be regenerated in the future.

Since knowledge of the mnemonic phrase allows you to immediately recover the BLS private keys, this phrase is kept secret. Therefore, it is imperative that the generated phrase is delivered to the user as securely as possible.

Currently, when the phrase is generated, it is output into the terminal so the user can copy it to use at a later time. However, the terminal buffer is not properly cleared. So for example, if a Linux user operates `gnome-terminal`, the mnemonic phrase is never cleared, and the phrase can be recovered later simply by scrolling up. This also applies to the macOS terminal and `iTerm2`.

Additionally, the clipboard is not blocked and is never cleared. Therefore, if a user copies this phrase from the terminal, it could remain in the clipboard for an indefinite period of time.

Exploit Scenario
Alice uses the Deposit CLI tool to generate her secret keys. An attacker, Eve, gains access to her device and steals her secret mnemonic phrase by extracting it from her terminal or clipboard. Alternatively, Alice visits a malicious website hosted by Eve which accesses Alice's clipboard and recovers the secret phrase.

Recommendation
Short term, properly clear the terminal buffer after the tool is no longer used. This will minimize the risk of users exposing sensitive information on-screen. The proper fix depends on the operating system and should be implemented carefully. In addition, clear the clipboard after the tool has finished or block the user from copying the data altogether.

Long term, consider presenting the user with the secret mnemonic phrase for only a short period of time (e.g., a few minutes), and then clearing this information upon timing out. This will minimize the risk of users exposing sensitive information onscreen.

2. Deposit stores a world-readable file with sensitive information

Severity: Low
Type: Data Exposure
Target: `eth2deposit/credentials.py`

Difficulty: High
Finding ID: TOB-EF-DEP-002

Description
The deposit tool saves the keys in world-readable files. This practice is risky and could lead to a compromise.

After generating the key, `deposit` stores the encrypted keys in a directory that is world-readable by default. Thus, any user with read access to a file's enclosing directory can access the private information. Also, note that world-readable (read-only) directory access is a common default on many Unix-based systems.

Exploit Scenario
An attacker learns that the deposit tool was installed on a machine, then gains local access to that machine. Since the default directory permissions are still in place, the attacker reads the keys from the saved files.

Recommendation
Short term, minimize the permissions when saving sensitive files. This will mitigate the risk in case an attacker accesses the system used for key generation.

Long term, review the amount of necessary permissions for each component of your system. This will mitigate risk in case an attacker accesses the system used for key generation.

3. Deposit does not provide entropy validation on passwords

Severity: Low
Type: Access Controls
Target: `eth2deposit/deposit.py`

Difficulty: Low
Finding ID: TOB-EF-DEP-003

Description
The Deposit CLI tool allows users to use weak or even empty passwords during key generation.

This tool is responsible for generating a BLS key pair. To protect the secret keys, the tool uses password-based key derivation functions to securely store the secrets. Therefore, tool users should provide a password that will be used to protect these secrets:

```
@click.password_option(prompt='Type the password that secures your validator keystore(s)')
def main(num_validators: int, mnemonic_language: str, folder: str, chain: str, password:
str) -> None:
   ...
   click.echo('Saving your keystore(s).')
   keystore_filefolders = credentials.export_keystores(password=password, folder=folder)
   click.echo('Creating your deposit(s).')
   deposits_file = credentials.export_deposit_data_json(folder=folder)
   click.echo('Verifying your keystore(s).')
   assert credentials.verify_keystores(keystore_filefolders=keystore_filefolders,
password=password)
   ...
```

Figure 3.1: *Password usage in the deposit tool.*

However, the Deposit CLI lacks user password validation while generating the mnemonic, which allows weak passwords to be used. This could allow an offline dictionary attack to happen.

Moreover, when calling the CLI with an optional argument `--password=''` users can generate a mnemonic with a null password. This command line argument is undocumented in the `ef-deposit-cli` documentation.

Exploit Scenario
Alice uses Deposit CLI to generate secret keys on her machine using a simple password. An attacker, Eve, tries to access the keystore by brute-forcing commonly used passwords and empty passwords. Once successful, Eve has access to the BLS signing key. This attack is

more feasible because Eve also has access to the sensitive keys stored in the saved files (as outlined in TOB-EF-DEP-002).

Recommendation
Short term, add a validation check on passwords to ensure minimum length. This will mitigate the risk of an attacker brute-forcing the encrypted keys.

Long term, use a password strength library/entropy calculator on user passwords to enforce use of strong passwords. This will mitigate the risk of an attacker brute-forcing the encrypted keys.

4. Saving large JSON integers could result in interoperability issues

Severity: Low
Type: Undefined Behavior
Target: `eth2deposit/credentials.py`

Difficulty: High
Finding ID: TOB-EF-DEP-004

Description
The parsing of `JSON` integers generated by the deposit tool differs from mainstream implementations such as `NodeJS` and `jq`.

The `JSON` standard warns about certain "interoperability problems" in numeric types outside the range `[-(2**53)+1, (2**53)-1]`. This issue is caused by some widely used `JSON` implementations employing `IEEE 754` (double precision) numbers to implement integer numbers.

For instance, if the deposit tool saves the following amount value `"1152921504606846976"` `(2**60)`, it is serialized as the expected value `1152921504606846976`. However, `NodeJS 10` and `jq 1.5` parse it as `1152921504606847000`.

```
[
  {
    ...
    "amount": 1152921504606846976,
    ...
  }
]
```

Figure 4.1: *Part of a* `JSON` *file produced by the deposit tool.*

Exploit Scenario
Alice uses the deposit tool to generate some `JSON` files. Later, some third-party software reads Alice's `JSON` but its interpretation of the number differs from the deposit tool's interpretation, causing unexpected behavior for Alice.

Recommendation
Short term, use strings instead of `JSON` numeric values to implement the amount field. This will prevent any ambiguity when parsing the numeric fields of the deposit `JSON` files.

Long term, provide a recommended `JSON` implementation to use when interacting with the `JSON` files generated by the deposit tool. This will prevent any ambiguity when parsing the numeric fields of the deposit `JSON` files.

References
- RFC 8259, Section 6: numbers in JSON

5. Use of `assert` will be removed when the bytecode is optimized

Severity: High
Type: Data Validation
Target: `eth2deposit`

Difficulty: High
Finding ID: TOB-EF-DEP-005

Description
The deposit tool uses `assert` to make sure the input for the tool is correct; however, such code will be removed when the bytecode is compiled with optimizations enabled.

The deposit tool contains several places where important verifications are performed using the `assert` statement:

```
@click.password_option(prompt='Type the password that secures your validator keystore(s)')
def main(num_validators: int, mnemonic_language: str, folder: str, chain: str, password:
str) -> None:
   ...
   click.echo('Saving your keystore(s).')
   keystore_filefolders = credentials.export_keystores(password=password, folder=folder)
   click.echo('Creating your deposit(s).')
   deposits_file = credentials.export_deposit_data_json(folder=folder)
   click.echo('Verifying your keystore(s).')
   assert credentials.verify_keystores(keystore_filefolders=keystore_filefolders,
password=password)
   click.echo('Verifying your deposit(s).')
   assert verify_deposit_data_json(deposits_file)
   click.echo('\nSuccess!\nYour keys can be found at: %s' % folder)
   click.pause('\n\nPress any key.')
```

Figure 5.1: *Part of the main function of the deposit tool.*

However, these checks will be removed if the bytecode is optimized using the `-O` flag, as indicated in the Python documentation:

```
       -O     Remove  assert  statements  and any code conditional on the value of
__debug__; augment the filename for compiled (bytecode) files by adding .opt-1 before
              the .pyc extension.
```

Figure 5.2: *Part of a Python3 documentation.*

It is also worth mentioning that the tool will fail to run in the optimized mode because the `asserts` removed in `path_to_nodes` function contain a side effect.

Exploit Scenario

Alice compiles the deposit tool with optimizations enabled. Later, she uses the tool, but the lack of validation produces invalid results without any visible fail.

Recommendation
Short term, replace the use of assertion for proper input validation and do not include assertions with side effects. This will enable the tool to run correctly even if the bytecode is optimized.

Long term, review every feature of the Python language you use in case it demonstrates different behavior when the bytecode is optimized. This will mitigate the risk of the tool running incorrectly if the bytecode is optimized

6. Passwords are accessible via shell history

Severity: Medium
Type: Access Controls
Target: `eth2deposit/deposit.py`

Difficulty: High
Finding ID: TOB-EF-DEP-006

Description
The Deposit CLI tool allows passwords to be supplied as command-line arguments, enabling password leakage through operating systems such as terminal shell history and execution of `ps` or similar commands.

Figure 6.1 shows how the tool allows specification of a password through command-line arguments:

```
$ python3 ./eth2deposit/deposit.py --password="mypassword"
```

Figure 6.1: *Running `deposit-cli` command by supplying password.*

However, when the shell history is accessed in a terminal window, the password is leaked in plaintext:

```
  516  python3 ./eth2deposit/deposit.py --password="mypassword"
  517  bash history
```

Figure 6.2: *Accessing shell history.*

Exploit Scenario
Alice uses the Deposit CLI to generate secret keys on her machine with a command-line argument `--password='12345'`. An attacker, Eve, gains access to Alice's machine, looks through previously run commands on the terminal, and finds the password used to secure the BLS signing key. In combination with TOB-EF-DEP-002, Eve also has access to the sensitive keys stored in the files.

Recommendation
Short term, do not allow users to use the command line argument to specify passwords. This will prevent the password from leaking to other users in the underlying operating system.

Long term, minimize the use of sensitive information and ensure secrets are not exposed via operating system logs to avoid leaking the sensitive information to other users.

7. PyInstaller binaries should be distributed with signatures

Severity: High
Type: Data Validation
Target: `eth2deposit`

Difficulty: High
Finding ID: TOB-EF-DEP-007

Description
The deposit tool is distributed as binary files compiled using PyInstaller, but without any way for users to verify their authenticity.

The deposit tool is compiled to a single binary using PyInstaller to be released on GitHub:

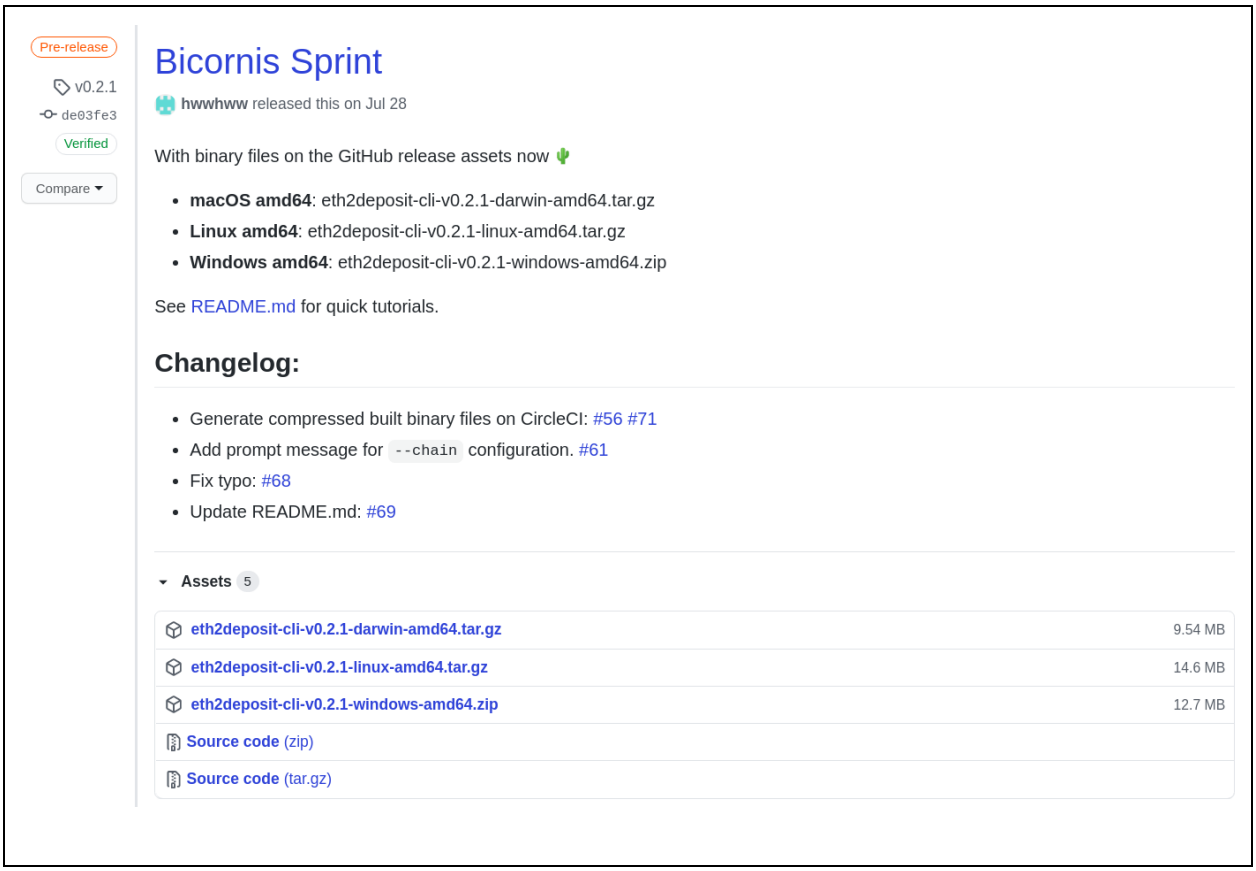

Figure 7.1: *Latest GitHub release.*

However, there is no way for users to check that these binaries are compiled by someone from the Ethereum 2.0 development team.

Exploit Scenario
Eve compromises the security of the GitHub account used to upload the deposit binaries, and replaces the binaries with backdoored versions. Alice downloads the released binaries and her keys are compromised.

Recommendation
Short term, include signatures to allow users to verify the released binaries. This will mitigate the risk of using malicious binaries if an Ethereum developer's Github account is compromised. We recommend using `gpg` with public keys uploaded to public servers.

Long term, review the chain of trust of the precompiled releases used to generate private keys. This will mitigate the risk of using malicious binaries if an Ethereum developer's GitHub account is compromised.

8. Certain encodings can make passwords impossible to input

Severity: Low
Type: Data Validation
Target: `eth2deposit`

Difficulty: High
Finding ID: TOB-EF-DEP-008

Description
The use of certain characters (e.g., ñ) in the password will produce different results according to the terminal encoding and how the password is provided.

The deposit tool was implemented to accept UTF-8 input, forcing the conversion to that encoding at each point of input handling:

```
    @staticmethod
    def _process_password(password: str) -> bytes:
        """
        Encode password as NFKD UTF-8 as per:
        https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2335.md#password-requirements
        """
        password = normalize('NFKD', password)
        password = ''.join(c for c in password if ord(c) not in UNICODE_CONTROL_CHARS)
        return password.encode('UTF-8')
```

Figure 8.1: `_process_password` *function to force UTF-8 encoding.*

However, it seems that the tool can still fail with the following list of errors when it is run by forcing `LANG="en_US-iso8859-1"`, `iso8859-1` as the terminal input encoding or the password specified in the command line:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position 0:
invalid continuation byte
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 0:
ordinal not in range(128)
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcf1' in
position 0: surrogates not allowed
```

Figure 8.2: *Different encoding errors.*

Additionally, Windows users could have issues since the use of UTF-8 in that operating system is not enabled by default.

Exploit Scenario
Alice generates a password with certain characters in one operating system. Later, she moves to another platform with unsupported encoding and is unable to re-enter her password.

Recommendation
Short term, validate the configuration of terminal encoding and system encoding to mitigate the risk of users employing configurations that fail to run the tool properly. Additionally, document the recommended configuration for users.

Long term, review the use of text encoding in each step of key generation and storage. This will mitigate the risk of users employing configurations that fail to run the tool properly. Also, properly document each corner case, especially for non-Latin languages.

9. Naming of the resulting JSON files can be misleading

Severity: Medium
Type: Data Validation
Target: `eth2deposit`

Difficulty: High
Finding ID: TOB-EF-DEP-009

Description
The naming convention of the output files depends on the current time and can confuse users if several keys are generated in the same directory.

The ETH2 deposit tool will ask the user for some information to generate an encrypted private key and deposit information. After the key and the deposit information are computed, the tool will save two `JSON` files—`keystore` and `deposit-data`—in an output folder. To create these files, the operating system's timestamp is included in the filename:

```
    def save_signing_keystore(self, password: str, folder: str) -> str:
        keystore = self.signing_keystore(password)
        filefolder = os.path.join(folder, 'keystore-%s-%i.json' % (keystore.path.replace('/',
'_'), time.time()))
        keystore.save(filefolder)
        return filefolder
```

Figure 9.1: `save_signing_keystore` *function.*

```
    def export_deposit_data_json(self, folder: str) -> str:
        deposit_data = [cred.deposit_datum_dict for cred in self.credentials]
        filefolder = os.path.join(folder, 'deposit_data-%i.json' % time.time())
        with open(filefolder, 'w') as f:
            json.dump(deposit_data, f, default=lambda x: x.hex())
        return filefolder
```

Figure 9.2: `export_deposit_data_json` *function.*

However, since the `time.time()` function is called twice, the names of the `JSON` files are not guaranteed to match. This could confuse users if they generate several keys in the same folder, which is the default option.

Exploit Scenario
Alice generates several validator keys with their corresponding deposit files. When uploading the deposit file to the ETH2 Launch Pad, she confuses the one she wants and uploads the incorrect one.

Recommendation
Short term, make sure the keystore and deposit file have always matching filenames so users won't be confused when selecting the deposit `JSON` file to upload.

Long term, make sure users have enough information to know exactly which deposit file they want to upload so users won't be confused when selecting the deposit `JSON` file to upload.

10. Python Crypto wrappings allow unsafe parameters

Severity: Informational
Type: Cryptography
Target: `utils/crypto.py`

Difficulty: N/A
Finding ID: TOB-EF-DEP-010

Description
The Deposit CLI tool uses a variety of cryptographic primitives, such as `scrypt`, `PBKDF2`, and `AES`. To include these primitives, the codebase provides wrapper functions around the Python Crypto library. These functions will, for example, take in an input password, salt, and other relevant parameters, and pass them into `scrypt`.

The password-based key derivation functions, `scrypt` and `PBKDF2`, are tunable by specific parameters (e.g., number of iterations for `PBKDF2`). These parameters can be adjusted so these functions are as slow or as fast as desired. However, if these parameters are set to low enough values, they can be considered unsafe for password storage. Currently, there are no restrictions on any of these parameters that are passed to both `scrypt` and `PBKDF2`.

The codebase defines `keystores` for both `scrypt` and `PBKDF2`, which select the parameters for the user, and these parameters are safe. However, it would be safer to wrap both `scrypt` and `PBKDF2` with some basic checks to ensure that the input parameters are not unsafe (e.g., enforcing a minimum number of iterations of 1,000).

Recommendation
Short term, determine which parameters are unsafe to ever use for `scrypt` and `PBKDF2`. Once these are determined, prevent unsafe use by adding a warning that checks if either `scrypt` or `PBDKF2` are given unsafe parameters.

Long term, consider adding checks that prevent both `scrypt` and `PBKDF2` from being called with unsafe parameters. Determining whether these parameters are unsafe depends on how these functions need to be used.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Specification

- **The checksum in `EIP 2335` is vulnerable to a length extension attack.** This currently does not seem to present any issues for the Deposit CLI implementation, but it might be good to mention this in the `EIP` to ensure those implementing this `EIP` are aware.
- **EIP 2335 does not recommend `scrypt` over `PBKDF2`.** Both primitives are presented as options but it may be beneficial to readers to emphasize that `scrypt` is a much stronger option.

Deposit

- **Validate the number of validators to disallow the use of zero or negative numbers.** Specifying negative or zero validators will crash or generate a key, but with an empty deposit `JSON`. Producing a clear error message will prevent confusion for users in case they try to employ an invalid number of validators.