NETTITUDE

NETTITUDE

A member of the Lloyd's Register group

Penetration Testing Technical Report

Prepared for: The Linux Foundation
System: Hyperledger Indy
Type: Security Assessment

Author: Graham Shaw
Date: 13 November 2018
Version: 1.0

PCI Security Standards Council ™

CREST

CHECK IT Health Check Service

1 Report Contents

2 Distribution List

Nettitude	Name	Title
	Graham Shaw	Security Consultant
	Jose Lopes	Security Consultant
	Miles Corn	Account Manager

The Linux Foundation	Name	Title
	David Huseby	Security Maven

3 Revision History

Version	Issue Date	Issued by	Comments
0.1	04 November 2018	Graham Shaw	Initial Draft
0.2	08 November 2018	Jose Lopes	Quality Assurance
0.3	09 November 2018	Miles Corn	Quality Assurance
1.0	13 November 2018	Graham Shaw	Final

4 Engagement Particulars

Background

This report serves as technical documentation for the recent penetration test performed for The Linux Foundation by Nettitude. For a high-level assessment of the tested environment, please refer to the associated management report:

```
MGMT_REPORT_Penetration_Test_The_Linux_Foundation_2018-10-31_v1.0.pdf
```

Rules of Engagement

The assessment was performed in line with the following rules of engagement:

- Nettitude's product assessment methodology was used.
- Testing of infrastructure owned by The Linux Foundation was not permitted.
- The testing and reporting was permitted and performed during a 43 day period; 19-Sep-18 to 31-Oct-18. Any results held in this report relate to the status of the tested environment on those dates.

Scope

The Linux Foundation tasked Nettitude to perform a security assessment with the following scope:

Component	Repository
Indy Node	https://github.com/hyperledger/indy-node
Plenum	https://github.com/hyperledger/indy-plenum
Hyperledger Indy Project Enhancements	https://github.com/hyperledger/indy-hipe
Shared crypto library	https://github.com/hyperledger/indy-crypto
Reference agents	https://github.com/hyperledger/indy-agent
Anonymous credentials	https://github.com/hyperledger/indy-anoncreds

Python wrapper test	https://github.com/hyperledger/indy-post-install-automation
Indy Jenkins Pipeline Library	https://github.com/hyperledger/indy-jenkins-pipeline-lib
Indy SDK	https://github.com/hyperledger/indy-sdk

Testing Windows Observations and Constraints

Hyperledger Indy has a large codebase, and it would be possible to spend a very large amount of time fruitfully looking for vulnerabilities. In this context, the time frame provisioned for the completion of this engagement represented a reasonable trade-off between time and thoroughness. No constraints were encountered during the engagement.

Findings Summary

During the engagement, a total number of nine findings were identified. The following table shows the categorisation by severity:

0	0	2	6	1
Critical	High	Medium	Low	Info.

5 Findings

1. Security Assessment

Component	Description	Severity	Recommendation	Ref.
indy-plenum	Use of *random.choice* to generate cryptographic seed	Medium	Rewrite *randomSeed* to use a secure random number generator.	6.1
product-wide issue	Sensitive data not consistently zeroed after use	Medium	Zero memory containing sensitive data that is no longer needed.	6.2
product-wide issue	Cryptographic operations do not execute in constant time	Low	Use cryptographic operations which execute in constant time.	6.3
indy-plenum	Class *storage.directory_store.DirectoryStore* potentially vulnerable to path traversal attack	Low	Either escape unsafe characters, or encode the entire key.	6.4
indy-plenum	Functions *ledger.util.count_bits_set* and *highest_bit_set* vulnerable to infinite loop	Low	Throw an exception if the input is negative.	6.5
indy-plenum	Race condition in *stp_zmq.util._create_file_with_mode*	Low	Use *os.open* in preference to open if a non-default file mode is required.	6.6

indy-plenum	Incorrect handling of insufficient data by *common.serializers.compact_serializer.CompactSerializer* for compound field names	Low	Check that items list is not empty before attempting to pop a value from it.	6.7
indy-plenum	Function *plenum.common.util.randomString* does not return fully random result	Low	Correct off-by-one error.	6.8
indy-plenum	*plenum.client.Wallet* is serialised using *jsonpickle*	Info.	Code already scheduled for removal, no further action required.	6.9

6 Analysis: Security Assessment

6.1 Medium: Use of *random.choice* to generate cryptographic seed

Description of the Issue

The function *stp_core.crypto.util.randomSeed* uses the function *random.choice* to generate the seed, but it is not suitable for cryptographic use.

According to the documentation, the values generated by the Python random module are generally pseudo-random in nature. It specifically warns that this makes them unsuitable for use in cryptographic applications:

> **Warning:** The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.

Figure 1: Warning in Python random module

In indy-plenum, the function *stp_core.crypto.util.randomSeed* uses *random.choice* to generate the seed:

```
def randomSeed(size=32):
    return ''.join(random.choice(string.hexdigits)
                   for _ in range(size)).encode()
```

Figure 2: Use of random.choice in function randomSeed

This is in turn called from a number of places, most of which appear to be ultimately for test purposes, however the call in *stp_zmq.util.createEncandSigKeys*:

```
def createEncAndSigKeys(enc_key_dir, sig_key_dir, name, seed=None):
    seed = seed or randomSeed()
```

Figure 3: use of function randomSeed in function createEncAndSigKeys

which is called from three other locations, and does not at first sight appear to be test code. Furthermore, even if all current usage were safe, there is no indication in *randomSeed* that it is unsuitable for cryptographic use. The same applies to functions which call it, directly or indirectly, that are not themselves obvious test harnesses.

Note that there are numerous examples of non-cryptographic random number generators being used in test code, however this is fully acceptable since it is not intended for use in a production environment.

Affected Components

- indy-plenum

Nettitude Recommends

The simplest option for remediation would be to:

1. Rewrite *randomSeed* to use a secure random number generator. A suitable choice for this, which is already used elsewhere by Indy, would be *libnacl.randombytes*.

Further Reading

- Python 3.7.1, random — Generate pseudo-random numbers (https://docs.python.org/3/library/random.html)

6.2 Medium: Sensitive data not consistently zeroed after use

Description of the Issue

Security would be improved by consistently zeroing sensitive information in memory when it is no longer needed.

In cryptographic software it is considered good practice to overwrite sensitive information with zeros once that information no longer needs to be kept. Under ideal circumstances this would not be necessary, as the operating system ought then to provide sufficient isolation between processes to ensure that no information can leak, regardless of how it is discarded. However this is not necessarily a safe assumption in practice, due to the potential for:

- Attacks against the operating system or CPU which circumvent process isolation. Notable examples include Meltdown and Spectre.
- Vulnerabilities in application or library code which allow leakage of information. An example of this is the Heartbleed vulnerability in OpenSSL.

Two of the main challenges which would face an attacker attempting to use these techniques are firstly performing the attack at a time when the sensitive information is present in memory in unencrypted form, and secondly finding the sensitive information from amongst what may be a very large volume of other data. Having more copies of the information present, for longer periods of time, is likely to make this easier.

However, whether this countermeasure is worthwhile in a particular context will depend upon the value of the sensitive information, the additional code complexity required to perform the zeroing (if it is feasible at all), and the likelihood of attack given the available attack surface.

Hyperledger Indy does zero some sensitive data after use, however it does not appear to do so routinely. For example, in the class *services::wallet::wallet::Keys*, the functions *serialize_encrypted* and *deserialize_encrypted* call the function *memzero* on the intermediate (serialized but unencrypted) values that they create.

```
let mut serialized = rmp_serde::to_vec(self)
    .map_err(|err| CommonError::InvalidState(format!("Cannot serialize keys: {:?}", err)))?;

let encrypted = encrypt_as_not_searchable(&serialized, master_key);

memzero(&mut serialized[..]);
```

Figure 4: Exampl of memzero being used

However, in the function *raw_master_key* in *services::wallet::encryption*, the intermediate value resulting from base58 decoding of the passphrase does not appear to be securely disposed of:

```
let bytes = &base58::decode(passphrase)?;
let key = chacha20poly1305_ietf::Key::from_slice(&bytes)
    .map_err(|_| ::errors::common::CommonError::InvalidStructure("Invalid Master Key length".to_string()))?;
Ok(key)
```

Figure 5: Example where sensitive data is not zeroed

Partial application of zeroing is of limited value, since the attacker only needs one copy of the sensitive data. However, a valid distinction can be drawn between the client and server components of Indy:

- These will typically be running in separate environments, so a weakness in one does not necessarily compromise the other.
- The server is arguably at lower risk, since it is likely to be run in a container in which it is the only substantive package executing.
- Individual server credentials are of limited value, since it is in the nature of a blockchain system not to trust individual nodes (however an attack which yielded credentials for a large part of the network would be of high value).

A further consideration is that zeroing would likely to be more straightforward for components written in Rust than in Python, due to the differing extent to which memory management is abstracted by these two languages. Options for obtaining better control over Python objects include use of mutable objects (such as bytearray) in place of immutable ones, or use of native code. Some care may be necessary to ensure that any zeroing operations are not optimized away by the compiler.

This issue has been classed as medium severity on account of the encrypted wallet capability, this being both at relatively high risk of attack, and having evidence that zeroing of memory was an intended design feature. For other parts of the codebase it could reasonably be considered low severity.

Affected Components

- Product-wide issue

Nettitude Recommends

1. It would be desirable for sensitive data to be routinely and consistently zeroed after use, throughout the codebase.
2. The SDK, and the encrypted wallet capability in particular, should be a higher priority in this respect than the server.

Further Reading

- Libsodium utility functions (https://docs.rs/sodiumoxide/0.0.16/sodiumoxide/utils/index.html)
- Clearing memory in Python (https://www.sjoerdlangkemper.nl/2016/06/09/clearing-memory-in-python/)

6.3 Low: Cryptographic operations do not execute in constant time

Description of the Issue

Some of the cryptographic operations performed by Hyperledger Indy have an execution time which depends on the specific values processed. This creates a risk of a timing attack being possible.

If the time taken to perform a cryptographic operation depends on the value of a private key or other sensitive data, then an attacker may be able to deduce the content of that data by observing how the elapsed time varies as a function of other input parameters.

In order to assess whether a timing attack was likely to be feasible, Nettitude simulated the behavior of the *mul* method of *pair::PointG1* in the libindy-crypto package. This uses *amcl::pair::g1mu*l to perform the underlying multiplication operation:

```
/// PointG1 ^ GroupOrderElement
pub fn mul(&self, e: &GroupOrderElement) -> Result<PointG1, IndyCryptoError> {
    let mut r = self.point;
    let mut bn = e.bn;
    Ok(PointG1 {
        point: g1mul(&mut r, &mut bn)
    })
}
```

Figure 6: Example of cryptographic operation which does not execute in constant time

The outcome was that:

- The execution time depended primarily on the magnitude of the multiplicand. The relationship appeared to be close to linear in the number of bits. This is unsurprising, and not necessarily a security concern.
- However, further dependency was observed on the number of bits set. This was of a lesser magnitude, typically of the order of 1%, but of significantly greater security concern due to it depending on the content rather than just the length.

On further investigation it was found that indy-crypto depends on version 0.1.2 of the amcl Rust bindings:

```
[dependencies]
amcl = { version = "0.1.2",  optional = true, default-features = false, features = ["BN254"]}
```

Figure 7: Declaration of amcl dependency to indy-crypto

which in turn depends on version 2.2 of the generic amcl library. In addition to being no longer supported, this precedes the introduction of constant-time algorithms in version 3.1:

```
Version 3.1 is a major "under the hood" upgrade. Field arithmetic is
performed using ideas from http://eprint.iacr.org/2017/437 to ensure
that critical calculations are performed in constant time. This strongly
```

Figure 8: Support for constant-time operations in amcl version 3.1

It therefore seems likely that this issue could be at partially addressed by utilizing a more recent version of amcl (however this is not the only cryptographic library on which Indy depends).

Servers are more naturally vulnerable to timing attacks than clients, however it is not inconceivable that (for example) and HTTP server might also be acting as an Indy client.

Affected Components

- Product-wide issue

Nettitude Recommends

1. Where practicable, use cryptographic operations which execute in constant time.

Further Reading

- David Brumley and Dan Boneh, Remote timing attacks are practical. USENIX Security Symposium, August 2003 (http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf)
- Paul C. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, CRYPTO 1996 (https://www.paulkocher.com/TimingAttacks.pdf)

6.4 Low: Class *DirectoryStore* potentially vulnerable to path traversal attack

Description of the Issue

If an attacker were able to access a *storage.directory_store.DirectoryStore* using arbitrary key names, it would be possible to access arbitrary files in the filesystem by means of a path traversal attack. No method of exploitation has been found for this, but nor has the possibility been excluded.

The *DirectoryStore* class implements a key-value store by mapping each key to a filename, and each value to the content of the corresponding file.

The method used to form the pathname for a key is simply to append it to a given base path, with an appropriate path separator between the two:

```
def keyFilePath(self, key: str):
    return os.path.join(self.dbPath, key)
```

Figure 9: Unsafe path construction in DirectoryStore

This does not protect against path traversal, specifically:

- Keys containing the path separator character '/'
- Keys containing '..' as a path component

Note in particular that if the key begins with a path separator, it is treated as an absolute path - causing it to completely replace the base path.

For this finding to be exploitable, it would be necessary for the attacker to access the key-value store using an arbitrary chosen key. Within the Indy codebase, the only usage of the *DirectoryStore* class would appear to be by *ClientReqRepStoreFile*. Keys are constructed from a *Request* object by combining the identifier and request ID fields with an intervening comma. Notwithstanding any validation which occurs elsewhere, for the purposes of a *Request* object:

- The identifier is supposed to be a string and can have arbitrary content.

- The request ID is supposed to be an integer, and there is a type hint to this effect, however in Python these are merely unenforced hints and there is nothing to prevent an arbitrary string from being used instead.

The inclusion of the comma in the key would be a serious inconvenience to an attacker, but not necessarily an insurmountable one.

A thorough audit of *Request* class usage has not been attempted, since this would take rather more effort than adding countermeasures to the *DirectoryStore* class, and in any event would not protect against future changes to the codebase.

Affected Components

- indy-plenum

Nettitude Recommends

Options for protecting against path traversal include:

- Providing an alternative representation for unsafe characters, introduced by some form of escape character or sequence.
- Transforming the key into a completely different representation using a safe (or safer) character set, for example using a variant of base-64.

The former is usually preferable when unsafe characters occur rarely (as will presumably be the case in this instance during normal use), the latter when they occur frequently.

Further Reading

- OWASP Project, Path Traversal (https://www.owasp.org/index.php/Path_Traversal)

6.5 Low: Functions *count_bits_set* and *highest_bit_set* can enter infinite loop

Description of the Issue

If a negative value is passed to *ledger.util.count_bits_set* or *ledger.util.highest_bit_set*, these functions will loop forever. Whilst they do not currently appear to be used to process any untrusted data, there would be a risk of a denial of service attack if they were so used in the future.

The function *count_bits_set* is built around a loop which is supposed to progressively reduce the input value to zero by clearing one bit per iteration:

```
def count_bits_set(i):
    # from https://wiki.python.org/moin/BitManipulation
    count = 0
    while i:
        i &= i - 1
        count += 1
    return count
```

Figure 10: Function count_bits_set

The same is true of the function *highest_bit_set*, although the mechanism differs in detail:

```
def highest_bit_set(i):
    # from https://wiki.python.org/moin/BitManipulation
    # but with 1-based indexing like in ffs(3) POSIX
    hi = i
    hiBit = 0
    while hi:
        hi >>= 1
        hiBit += 1
    return hiBit
```

Figure 11: Function highest_bit_set

The operation *i &= i - 1* has the effect of clearing the least significant bit. It works for fixed-length twos-complement integers (provided that overflows are tolerated), but not for the arbitrary-length integers provided by Python.

The operation *hi >>= 1* causes *hi* to be divided by 2, then rounded towards minus infinity. This has the effect of always making the magnitude of *hi* smaller if it is positive, but larger if it is negative.

Affected Components

- indy-plenum

Nettitude Recommends

The safest course of action would be to modify the functions so that negative inputs have well-defined behavior. Since integers in Python are of arbitrary length, the appropriate course of action would likely be to throw an exception.

Alternatively, if the above is not acceptable, it would be reasonable to add a comment to the effect that it is the caller's responsibility to ensure that the input is non-negative, and that the behavior in the case of negative inputs is undefined.

6.6 Low: Race condition in *_create_file_with_mode*

Description of the Issue

There is a race condition in the function *stp_zmq.util._create_file_with_mode* which could potentially allow another user to gain access to the file content.

The function *stp_zmq.util._create_file_with_mode* is called to create files for storing public or private keys, for example by the function *_write_secret_key_file*:

```
def _write_secret_key_file(key_filename, current_time,
                           public_key, secret_key, metadata):
    banner = _cert_secret_banner.format(current_time)
    _create_file_with_mode(key_filename, 0o600)
    _write_key_file(key_filename,
                    banner,
                    public_key,
                    secret_key=secret_key,
                    metadata=metadata,
                    encoding='utf-8')
```

Figure 12: Example of usage of _create_file_with_mode

In this instance, the file in question will contain a secret key and it must not therefore be accessible to other users. That is achieved by requesting a file mode of 0600.

The function *_create_file_with_mode* performs its task in two steps: first it creates the file with the default access mode, then it changes the access mode to the one requested:

```
def _create_file_with_mode(path, mode):
    open(path, 'a').close()
    os.chmod(path, mode)
```

Figure 13: Method _create_file_with_mode

On POSIX-compatible systems it is usual for the default file access mode to at least allow read access from users in the same primary group, and not unusual for it to make files world-readable. This creates a short window of opportunity during which an attacker could open the file for reading. Once successfully opened by an attacker, and

provided the attacker keeps it open, changes to the access mode would not protect the content of the file from being read.

Affected Components

- indy-plenum

Nettitude Recommends

If non-default file permissions are required then use the python function *os.open* in preference to open. On POSIX-based systems this can be expected to map to the corresponding system function, which is supposed to act atomically upon the filesystem. Whilst this should not be considered an absolute guarantee of secure behavior, it is likely to behave as intended for local native filesystems, and in the worst case should at least be an improvement on two separate system calls. Ideally *O_EXCL* would be used in combination with *O_CREAT*, with pre-existing files handled separately (if at all).

Further Reading

- The Open Group Base Specifications Issue 7, "open" (http://pubs.opengroup.org/onlinepubs/9699919799/functions/open.html)

6.7 Low: Compound field name handling by *CompactSerializer*

Description of the Issue

When there is insufficient data for *CompactSerializer* to deserialize a field with a compound name, an exception is thrown instead of setting the field to *None*. If this class was used to deserialized untrusted data, that might conceivably permit a denial of service attack.

The usual behavior of *CompactSerializer* should there be insufficient data when deserializing is to set any missing fields to *None*. There is documentation to this effect in the implementation:

```
# if we have more fields than data available, assume that all
# missed fields are None
result[name] = None
```

Figure 14: Handling of insufficient data for non-compound names

This ensures that such fields can be accessed without first checking that the key exists (or alternatively, taking the risk of throwing an exception of type *KeyError*).

Fields with compound names are handled separately from non-compound names. The latter includes a check that there are input items remaining before deserializing:

```
elif items:
    result[name] = self._destringify(name, items.pop(0), fields)
```

Figure 15: Test for sufficient data remaining for non-compound names

whereas the former does not:

```
if "." in name:
    nameParts = name.split(".")
    ref = result
    for part in nameParts[:-1]:
        if part not in ref:
            ref[part] = {}
        ref = ref[part]
    ref[nameParts[-1]
        ] = self._destringify(name, items.pop(0), fields)
```

Figure 16: Handling of compound names

The outcome is that the *deserialize* method can be induced by the data it reads to throw an *IndexError* when this should not be possible. Whether this could be leveraged to create a usable security exploit is questionable, and it is borderline whether it should be considered a security issue. However it is at least a correctness issue, and should be a straightforward matter to fix.

Affected Components

- indy-plenum

Nettitude Recommends

1 Modify deserialization function to check that items list is non-empty before attempting to pop a value from it.

6.8 Low: Function *randomString* does not return fully random result

Description of the Issue

There is an off-by-one error in the function *plenum.common.util.randomString* which means that returned values containing an odd number of hex digits cannot end with an 'F'.

This occurs because the final digit is obtained by evaluating the expression *randombytes_uniform(15)*. The intent was presumably to obtain a number between 0 and 15 inclusive, however the function interprets its argument as an exclusive upper bound, therefore the value 15 can never be generated.

To test this, Nettitude generated 100,000 values using *randomString* each with 3 hex digits, then counted the number of occurrences of each value. As expected, no results ending in an 'F' were observed.

The effect is to reduce the entropy of generated values by approximately 0.1 bits compared to a random number generator with a uniform distribution. It is very unlikely that this would ever make the difference between a vulnerability being exploitable or not, however a fix is recommended on the grounds that the code does not behave as expected.

(The term ‘bit’ is used here in its sense as a unit of information. The entropy of a random number generator in bits is equal to $-\log_2 p$, where p is the probability that a particular value will be generated on a particular occasion. An alternate way of quantifying the difference is that it reduces the time required for a brute force attack by 6.25%.)

Affected Components

- indy-plenum

Nettitude Recommends

1 Fix off-by-one error to allow all 16 possible digits to be generated.

6.9 Informational: *plenum.client.Wallet* is serialised using *jsonpickle*

Description of the Issue

The (deprecated) class *plenum.client.Wallet* is serialized using *jsonpickle*, which could be exploited to enable arbitrary code execution in code managing a wallet for a third party.

Note that the class to which this finding relates, plenum.client.Wallet, had already been deprecated prior to the start of this security assessment. The risk described here should therefore only materialize in the event that there is legacy code using this class in the particular manner described, which is thought to be unlikely.

The *jsonpickle* website contains the following warning:

> **Warning:** jsonpickle can execute arbitrary Python code. Do not load jsonpickles from untrusted / unauthenticated sources.

Figure 17: Warning in jsonpickle documentation

(*jsonpickle* is not unusual in this regard: deserialisation of untrusted data is inherently risky, especially in a weakly-typed language such as Python.)

Deserialisation is performed by the decrypt function in *plenum.client.Wallet*:

```
def decrypt(ec: EncryptedWallet, key: bytes) -> 'Wallet':
    decryped = crypto_secretbox_open(ec.raw, ec.nonce, key)
    decoded = decryped.decode()
    wallet = jsonpickle.decode(decoded)
    return wallet
```

Figure 18: Deserialization of encrypted wallet

In typical usage, it would be reasonable to expect wallets to be trusted data due to the credentials that they contain. An attacker with control over an unencrypted wallet already has the ability to impersonate the corresponding user, and it can be argued that code execution adds little to this. However, it is not inconceivable that a service could be created to manage wallets on behalf of third parties. Under those circumstances, it might be advantageous for an attacker to create a malicious wallet in order to achieve code execution on the server.

Affected Components

- indy-plenum

Nettitude Recommends

The Linux Foundation has indicated that this code is deprecated and already scheduled for removal. No further action is required.

Further Reading

OWASP Top 10 (2017) A8: Insecure Deserialization (https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization)

7 Appendix

A. Severity Rating Matrix

The severity rating is determined by the likelihood and impact of a vulnerability on a system and, where possible, in the context in which that vulnerability is exposed, e.g. remote attack vs. internal attack.

The table below is used to calculate the overall severity rating of an issue based on these criteria.

This is not an assessment of risk as it does not include a valuation of the data or system, but it does provide the ability to prioritize the vulnerabilities identified within the target system or application and to integrate into their own risk management systems.

Likelihood \ Impact	Negligible	Minimal	Moderate	Major	Catastrophic
Rare	LOW	LOW	LOW	MEDIUM	HIGH
Unlikely	LOW	LOW	MEDIUM	HIGH	CRITICAL
Moderate	LOW	MEDIUM	MEDIUM	HIGH	CRITICAL
Likely	MEDIUM	MEDIUM	HIGH	CRITICAL	CRITICAL
Very Likely	MEDIUM	HIGH	HIGH	CRITICAL	CRITICAL

Likelihood

The likelihood rating of a vulnerability encompasses both the likelihood of the vulnerability being identified and attacked as well as the likelihood of that attack being successful. This is evaluated by taking into consideration the following elements:

Exploitability

- Difficulty and technical knowledge or skill required to identify/exploit the issue
- Time or resources required to mount a successful attack
- Availability of exploit code and automated attack tools

Reproducibility

- Ease of reproducing a successful attack

- Additional requirements for the attack to be successful, for example:
  - Victim user must be logged in
  - Some level of interaction by the victim user is required

Discoverability

- Number of instances of the vulnerability identified in the system
- Level of authentication required to access affected components
- Accessibility of the system (internet-facing or internal)
- Degree of specific Insider knowledge required

Frequency

- How often the issue is likely to occur over a period of time
- History of the issue in the industry
- Existence of self-propagating malware targeting the issue

These factors will be employed to formulate a final likelihood rating for a given issue.

Impact

The impact rating assesses the significance of exposure to a particular vulnerability. This is evaluated by considering the impacts to the affected system and the underlying business. The factors under consideration are outlined in the following table.

Impact	Negligible	Minimal	Moderate	Major	Catastrophic
Confidentiality	Disclosure of public information	Minor disclosure of commercial-in-confidence information	Major disclosure of commercial-in-confidence information	Minor disclosure of highly-confidential information	Major disclosure of highly confidential information
Integrity	Unauthorized modification of public data	Small-scale unauthorized modification of private data	Large-scale unauthorized modification of private data	Small-scale unauthorized modification of trusted data	Large-scale unauthorized modification of trusted data
Availability	Minor increase in processing load	Minor outage in a business system	Outage or unavailability of a business system	Extended unavailability or outage of a business system	Unavailability or outage of a business-critical system
Brand or Reputation	Complaints from small number of customers	Complaints from small number of customers across a broader customer base	Complaints from a large number of customers and localized media coverage	Short term adverse large scale media coverage	Extended adverse large scale media coverage
Regulatory and Legal	Warnings for minor breaches	Formal caution for regulatory breaches or threat of legal proceedings	Targeted audit / investigation by regulator or minor legal proceedings brought against the organization	Fines imposed and negative media coverage or major legal proceedings brought against the organization	Service line closed down

B. Penetration Testing Methodology

Nettitude has a series of approaches for conducting Penetration Tests.

Black Box Testing

In a Black Box test, the client does not provide Nettitude with any information about their infrastructure. For internal tests the customer may provide no more than a network point for the tester to connect in to. For external tests, this may simply be a URL or even just the company name that is in scope for assessment.

Nettitude is tasked with testing the environment as if they were an attacker with no information about the infrastructure or application logic that they are testing. Black Box tests tend to take longer to commission than White Box tests and may identify less exposures and vulnerabilities than those of White Box tests.

White Box Testing

In a White Box test, clients provide Nettitude with information about the applications and infrastructure prior to the commencement of the testing engagement. Usernames and Passwords are provided to Nettitude's testing team as part of the engagement, and the client may provide Nettitude’s consultants with access to source code. In this type of testing engagement, Nettitude works closely with the client to perform the assessment. These types of tests tend to gain deeper understanding of the application and infrastructure logic, and may generate highly comprehensive test results.

Grey Box Testing

A Grey Box test is a blend of Black Box testing techniques and White Box testing techniques. In Grey Box testing, clients provide Nettitude with snippets of information to help with the testing procedures. This results in a highly focused test.