Mina Client SDK, Signature Library and Base Components – Cryptography and Implementation Review

O(1) Labs Operating Co.

February 21, 2022 – Version 1.0

Prepared for

Emre Tekisalp
Izaak Meckler
Aneesha Raines
Bijan Shahrokhi

Prepared by

Eric Schorn
Ava Howell

Executive Summary

Synopsis

During October 2021, O(1) Labs engaged NCC Group's Cryptography Services team to conduct a cryptography and implementation review of selected components within the main source code repository for the Mina project. Mina implements a cryptocurrency with a lightweight and constant-sized blockchain, where the code is primarily written in OCaml. The selected components involved the client SDK, private/public key functionality, Schnorr signature logic and several other related functions. Full access to source code was provided with support over Discord, and two consultants delivered the engagement with eight person-days of effort.

Scope

The project scope centered around the main Mina repository commit `a1d868f` and included three central elements and supporting documentation:

- client_sdk
  - src/app/client_sdk/*
- mina_base
  - src/lib/mina_base/signed_command_payload.ml
  - src/lib/mina_base/signed_command.ml
- Documentation from within the repository and also at:
  - https://docs.minaprotocol.com/en
  - https://eprint.iacr.org/2020/352
- signature_lib
  - src/nonconsensus/signature_lib/private_key.ml
  - src/nonconsensus/signature_lib/schnorr.ml
  - src/nonconsensus/signature_lib/public_key.ml
  - src/nonconsensus/rosetta_coding/coding.ml
  - src/nonconsensus/snark_params/snark_params_nonconsensus.ml

The source code files are considered the 'entry points' and execution flow was traced throughout the repository.

Limitations

While the in-scope functionality lies within a much larger system context, good coverage was achieved over all in-scope material. Execution flow was traced through the client_sdk JavaScript code, but no other non-OCaml paths (e.g. Rust, Go or C).

Key Findings

The in-scope code was carefully architected, conservatively implemented, and generally exhibited a high degree of quality. However, the review did uncover a set of common application flaws involving:

- **Incomplete Bounds Checking on Random Private Key:** The code may not detect randomness sourced from a broken, intermittently failing or low-entropy random generator, and cannot generate a very small proportion of legal private key values.
- **Missing Private Key Validation:** The code may allow multiple base58check encoded private keys to map to the same decoded value which risks confusion, downstream malleability and/or interoperability issues.
- **Missing Bounds Validation of Signature Values:** The code may allow multiple signatures to be successfully verified against the same message, and thus lead to malleability concerns.
- **Outdated Dependencies and Build Warnings:** The code may allow an attacker to identify and utilize vulnerabilities in outdated dependencies to exploit the application.

As can be seen above, the majority of identified issues involved input validation. Note that the documentation (and testing) did not prominently specify detailed encoding formats or constraints.

Strategic Recommendations

NCC Group recommends addressing the findings from this engagement and prioritizing several aspects of future development as follows:

- Precisely define and prominently document exact encoding formats and constraints.
- Review opportunities for input validation and implement aggressive reject policies.
- Develop additional test cases to include negative code paths, particularly around input validation.

An ongoing program of incremental-testing of incremental-implementation will support a robust code base.

Dashboard

Target Metadata

Name	Mina Repository
Type	Selected components in/around client SDK
Platforms	Ocaml with C/Rust bindings
Environment	Testing

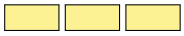
Engagement Data

Type	Code Review
Method	Manual source code analysis
Dates	2021-10-11 to 2021-10-15
Consultants	2
Level of Effort	8 person-days

Targets

Commit `a1d868f` of https://github.com/MinaProtocol/mina

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	2	
Low issues	3	
Informational issues	1	
Total issues	**6**	

Category Breakdown

Cryptography	4	
Data Validation	1	
Patching	1	

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 14.

Title	ID	Risk
Missing Private Key Validation	002	Medium
Missing Bounds Validation of Signature Values	004	Medium
Incomplete Bounds Checking on Random Private Key	001	Low
Outdated Dependencies and Build Warnings	003	Low
Issues Involving Unicode, UTF-8/16, ASCII, JavaScript, and Ocaml	005	Low
Lack of Domain Separation Tags in Random Oracles	006	Informational

Finding Missing Private Key Validation

Risk **Medium** Impact: Medium, Exploitability: Medium

Identifier NCC-E002674-002

Category Cryptography

Location
- The `publicKeyOfPrivateKey` method on lines 22-29 of src/app/client_sdk/client_sdk.ml
  - Within similar functions that consume a base58check encoded private key such as `rawPublicKeyOfPrivateKey`, `validKeypair`, `signString`, `signPayment`, `signStakeDelegation` and `signRosettaTransaction`
- The `of_base58_check_exn` function on lines 131-133 of src/lib/signature_lib/private_key.ml
- The `decode_exn` function on lines 58-81 of src/lib/base58_check/base58_check.ml

Impact Without checking a required base58check encoded string length or constraining the handling leading zero values, multiple encoded private keys will map to the same value. This is exacerbated by a missing range check on the decoded value. This may result in confusion, downstream malleability and/or interoperability issues.

Description In the context of elliptic curves, private keys are typically a non-zero scalar modulo the curve order.[1] An injective string:key encoding is desired and the correct range must be validated to satisfy downstream assumptions.

The `mina` source file `client_sdk.ml` provides the method `publicKeyOfPrivateKey` implemented on lines 22-29 which calculates the public key that corresponds to a supplied private key. Early in the process (line 25), the code utilizes the `Private_key.of_base58_check_exn` function. This function is implemented in `private_key.ml` and in turn utilizes `Base58_check.decode_exn` to perform the checked decoding of a base58 string. This latter function is excerpted below.

```
58 let decode_exn s =
59   let bytes = Bytes.of_string s in
60   let decoded =
61     try B58.decode mina_alphabet bytes |> Bytes.to_string
62     with B58.Invalid_base58_character ->
63       raise (Invalid_base58_character M.description)
64   in
65   let len = String.length decoded in
66   (* input must be at least as long as the version byte and checksum *)
67   if len < version_len + checksum_len then
68     raise (Invalid_base58_check_length M.description) ;
69   let checksum =
70     String.sub decoded
71       ~pos:(String.length decoded - checksum_len)
72       ~len:checksum_len
73   in
74   let payload =
75     String.sub decoded ~pos:1 ~len:(len - version_len - checksum_len)
76   in
77   if not (String.equal checksum (compute_checksum payload)) then
78     raise (Invalid_base58_checksum M.description) ;
79   if not (Char.equal decoded.[0] version_byte) then
80     raise (Invalid_base58_version_byte (decoded.[0], M.description)) ;
81   payload
```

[1] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-04#section-2.3

As can be seen above, the code correctly throws exceptions for invalid characters on line 63, a string shorter than (effectively) 5 on line 68, a bad checksum on line 78, and a bad version byte on line 80.

However, the expected string length of an encoded public key is not specified nor checked. Note that leading zeros (which are encoded as `1`s) have no special handling logic regarding occurrence or repetition. These statements also apply to the described enclosing code. As a result, many different base58check encoded strings will decode to exactly the same private key numeric value.

In addition, there is no bounds checking performed to ensure the private key lies within the `[1,r-1]` (inclusive) range where `r` is the related curve order. This may indeed be happening within the code behind the C/Rust bindings, but this would be obscure and extremely brittle. While the (sibling) encoding process has a concept of `max_encodable_length` the decoding process does not. Thus, if there is no downstream reduction taking place, an (egregiously) oversized string may cause excess resource consumption during decoding or during subsequent arithmetic operations.

Recommendation

Validate an exact required string length or disallow leading zeros. In the latter case, set and validate a maximum input string length.

Validate that the private key lies within `[1, r-1]` inclusive.

The fix should likely reside within the `of_base58_check_exn` function implemented within `private_key.ml` to cover all the 'entry' points in the first noted location above.

Finding **Missing Bounds Validation of Signature Values**

Risk **Medium** Impact: Medium, Exploitability: High

Identifier NCC-E002674-004

Category Cryptography

Location The `verify` function on lines 227-245 of src/lib/signature_lib/schnorr.ml

Impact A missing check on the bounds of the `r` and `s` signature values may allow multiple signatures to be successfully verified against the same message, and thus lead to malleability concerns.

Description Schnorr signatures typically consist of an `r` integer within the `[0, p-1]` (inclusive) range and an `s` integer within the `[0, n-1]` (inclusive) range, where `p` is the field characteristic and `n` is the curve order.[2] The values must be validated to lie within their relevant range to prevent malleability issues, e.g., downstream logic may/will calculate the same result for $s \cdot G$ given both `s` and `s+n`.

The `verify` function implemented in `schnorr.ml` is excerpted below. The `r` and `s` parameters can be seen in the function declaration on line 227, a hashing mode selection takes place within a `match` clause over lines 231-237, and the `r` parameter is provided directly to the hash function on line 239 without having its value validated. The scenario for the `s` parameter is similar with the calculation of $s \cdot G$ highlighted on line 240.

```
227 let verify ?signature_kind ((r, s) : Signature.t) (pk : Public_key.t)
228     (m : Message.t) =
229   let hash =
230     let open Mina_signature_kind in
231     match signature_kind with
232     | None ->
233         Message.hash
234     | Some Mainnet ->
235         Message.hash_for_mainnet
236     | Some Testnet ->
237         Message.hash_for_testnet
238   in
239   let e = hash ~public_key:pk ~r m in
240   let r_pt = Curve.(scale one s + negate (scale pk e)) in
241   match Curve.to_affine_exn r_pt with
242   | rx, ry ->
243       is_even ry && Field.equal rx r
244   | exception _ ->
245       false
```

Note that the encoding of `s` and `r` is sufficient to include multiples of the Pasta field characteristic and curve order values. Thus, all signatures become malleable. For completeness, a brief technical exchange took place over Slack that included the following note:

> @ihm: Ok- just checked. In the js build no bounds checks are performed but all arithmetic operations still function correctly

Recommendation Validate that `r` lies within `[0, p-1]` (inclusive) and `s` lies within `[0, n-1]` (inclusive), where `p` is the field characteristic and `n` is the curve order.

[2] https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki#Verification

Finding **Incomplete Bounds Checking on Random Private Key**

Risk **Low** Impact: Medium, Exploitability: Low

Identifier NCC-E002674-001

Category Cryptography

Location
- The `genKeys` method on lines 32-40 of src/app/client_sdk/client_sdk.ml
- The `create` function on lines 77-114 of src/lib/signature_lib/private_key.ml

Impact A random and typically illegal private key of all zeros, which may be sourced from a broken, intermittently failing or low-entropy random generator, will not be detected. Further, a very small proportion of private key values are not reachable.

Description In the context of elliptic curves, private keys are typically a non-zero scalar modulo the curve order.[3] The correct range must be validated to satisfy downstream assumptions.

The `mina` source file `client_sdk.ml` provides the method `genKeys` implemented on lines 32-40 which generates a new private key along with its public key counterpart. The process begins with a call to the `create` function implemented on lines 77-114 of `private_key.ml` as excerpted below.

```
77  let create () : t =
78    let open Js_of_ocaml in
79    let random_bytes_32 =
80      Js.Unsafe.js_expr
81        {js|(function() {
82          var topLevel = (typeof self==='object' && self.self===self && self) ||
83            (typeof global === 'object' && global.global === global && global) ||
84            this;
85          var b;
86
87          if (topLevel.crypto && topLevel.crypto.getRandomValues) {
88            b = new Uint8Array(32);
89            topLevel.crypto.getRandomValues(b);
90          } else {
91            if (typeof require === 'function') {
92              var crypto = require('crypto');
93              if (!crypto) {
94                throw 'random values not available'
95              }
96              b = crypto.randomBytes(32);
97            } else {
98              throw 'random values not available'
99            }
100         }
101         var res = [];
102         for (var i = 0; i < 32; ++i) {
103           res.push(b[i]);
104         }
105         res[31] &= 0x3f;
106         return res;
107       })|js}
108   in
109   let x : int Js.js_array Js.t = Js.Unsafe.fun_call random_bytes_32 [||] in
110   let byte_undefined () = failwith "byte undefined" in
```

[3] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-04#section-2.3

```
111  Snarkette.Pasta.Fq.of_bigint
112    (Snarkette.Nat.of_bytes
113       (String.init 32 ~f:(fun i ->
114            Char.of_int_exn (Js.Optdef.get (Js.array_get x i) byte_undefined))))
```

Random bytes are obtained on line 96 shown above. No subsequent check is made to disallow all zeros. While the all-zeros case is cryptographically unlikely and so the check may be perceived to be unnecessary, the industry provides multiple examples of low-entropy, intermittently failing or broken random number generators delivering fixed or otherwise invalid values.[4,5]

The logical AND implemented on line 105 above serves to limit the maximum value of the random number (destined for the private key) such that the most significant digit is less than or equal to `0x3f`. However, the Pasta field-characteristics[6] (or curve-orders) are slightly beyond this range, notably:

`0x40000000000000000000000000000000224698fc094cf91b992d30ed00000001` and `0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001`

As a result, the code is unable to produce the full (upper) range of public keys. Note that the unreachable portion is exceedingly small in relation to the full range.

Recommendation Ensure the `create` function is unable to return a zero for the private key; validate the random bytes against the all-zero case.

Consider a more precise constraint covering the entire scalar range, e.g., perhaps involving a Big Integer reduction modulo the curve order. Refer to the definition of Full Entropy on page 4 of NIST SP 800 90c.[7]

[4] https://arstechnica.com/gadgets/2019/10/how-a-months-old-amd-microcode-bug-destroyed-my-weekend/
[5] https://nvd.nist.gov/vuln/detail/CVE-2020-6616
[6] https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/
[7] https://csrc.nist.gov/csrc/media/publications/sp/800-90c/draft/documents/draft-sp800-90c.pdf

Finding	**Outdated Dependencies and Build Warnings**
Risk	**Low** Impact: Undetermined, Exploitability: Low
Identifier	NCC-E002674-003
Category	Patching
Location	• src/opam.export • src/app/libp2p_helper/src/go.mod
Impact	An attacker may attempt to identify and utilize vulnerabilities in outdated dependencies to exploit the application.

Description	Using outdated dependencies with discovered vulnerabilities is one of the most common and serious route of application exploitation. Many of the most severe breaches have relied upon exploiting known vulnerabilities in dependencies.[8]
	
	While overall dependency inspection is marginally outside of the project scope, the following message was noticed during the initial build process:
	
	`$ opam switch import src/opam.export`
	
	`<><> conf-openssl.1 installed successfully ><><><><><><><><><><><><><><><><><><><>`
	`=> This package shouldn't be used. Please use conf-libssl.1 or conf-openssl.2 instead.`
	
	A brief sampling of `src/opam.export` suggests a number of OpenSSL and field related dependencies may be outdated, including:
	
	• `async_ssl.v0.13.0`
	• `conf-libssl.2`
	• `conf-openssl.1`
	• `fieldslib.v0.13.0`
	
	In addition, the `go.mod` file noted above indicates Golang version 1.13. Golang is on a roughly 6-month release cycle with the latest version 1.17 delivered in August 2021.[9] The interim releases have introduced meaningful changes to the language, tools, runtime and core library, including a command line flag to enable Spectre mitigations.[10] Go Ethereum has disclosed[11] a critical DoS-related security vulnerability in Golang versions prior to 1.15.5[12] and 1.14.12 as CVE-2020-28362.[13]
	
	Finally, the build process is very complex and produces a large number of warnings. The large amount of output may obscure true issues.
	
	This finding is reported for completeness and caution.

Recommendation	Update all project dependencies. Set a periodic gating milestone for reviewing dependencies. Simplify the build process and aim to minimize the emitted warnings.

[8] https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/
[9] https://golang.org/doc/go1.17
[10] https://golang.org/doc/go1.15#compiler
[11] https://github.com/ethereum/go-ethereum/security/advisories/GHSA-m6gx-rhvj-fh52
[12] https://groups.google.com/g/golang-announce/c/NpBGTTmKzpM
[13] https://www.cvedetails.com/cve-details.php?cve_id=CVE-2020-28362

Finding **Issues Involving Unicode, UTF-8/16, ASCII, JavaScript, and Ocaml**

Risk **Low** Impact: Low, Exploitability: Medium

Identifier NCC-E002674-005

Category Data Validation

Location
- Most likely in src/app/client_sdk/client_sdk.ml and src/app/client_sdk/js_util.ml related functionality.
- For example, the `string_bits` function on lines 52-54 of src/app/client_sdk/string_sign.ml

Impact Discontinuities related to Unicode encoding and normalization involving UTF-8/16 as handled by JavaScript and Ocaml can lead to interoperability and string malleability issues.

Description Unicode has superseded ASCII as the standard character encoding for textual information and is typically transferred between applications as UTF-8 encoded byte streams. As the first 128 Unicode code points (and their UTF-8 encodings) map exactly onto 'identical' ASCII encodings, dissimilarities are typically not an issue for standard English characters/text. This includes values that are encoded in base58check, which has a very constrained character set, as consumed in `client_sdk/*` functionality. However, functions such as `signString`[14] in `client_sdk.ml` which is intended to sign an arbitrary string and `payload_common_of_js`[15] in `js_util.ml` which extracts a freeform memo string field, must consider various character encoding discontinuities unless their character set is tightly constrained to the lower-order ASCII set. There are two categories of issues to consider: 1) encoding widths, 2) Unicode normalization.

1. Encoding widths Early versions of Unicode contained less than 2^{16} code points (characters), so several languages adopted a 16-bit character type, including both Java[16] and JavaScript.[17] More recent versions of Unicode now contain over 1M code points so have exceeded this limit. As a result, some code points must be encoded as multiple (two) characters. This can introduce issues over the definition of string length as well as string indexing (e.g. bytes vs chars).

Unicode is typically encoded in variable-width UTF-8 for transport as shown below (diagram from Wikipedia). As can be seen, the first 128 code points align nicely with ASCII. Ocaml considers strings to be arbitrary sequences of bytes, so they can hold any kind of textual encoding. However, the recommended encoding for storing Unicode text in OCaml strings is UTF-8.[18]

Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

[14] https://github.com/MinaProtocol/mina/blob/a1d868f303b88e4d28faeddcd76e95314d978332/src/app/client_sdk/client_sdk.ml#L97-L102
[15] https://github.com/MinaProtocol/mina/blob/a1d868f303b88e4d28faeddcd76e95314d978332/src/app/client_sdk/js_util.ml#L29-L44
[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Character.html
[17] https://tc39.es/ecma262/multipage/ecmascript-data-types-and-values.html#sec-ecmascript-language-types-string-type
[18] https://ocaml.org/api/String.html

In the context of Mina's `client_sdk`, (for example) an external JSON message arrives encoded in UTF-8. This then ultimately becomes a JavaScript string in UTF-16. The latter string is delivered to the `Js.to_string`[19] function supplied by `js_of_ocaml`. The source code of this function makes a number of design decisions to handle the variable widths involved while transcoding back to Ocaml and UTF-8.[20]

2. Unicode normalization When entering the same logical string, differently encoded Unicode strings may arise from malicious intent or simply the broad range of participating devices, operating systems, locales, languages and applications involved.

Divergent string encoding typically involves characters with accents or other modifiers that have multiple correct Unicode encodings. For example, the Á (a-acute) glyph can be encoded as a single character U+00C1 (the "composed" form) or as two separate characters U+0041 then U+0301 (the "decomposed" form). In some cases, the order of a glyph's combining elements is significant and in other cases different orders must be considered equivalent.[21] At the extreme, the character U+FDFA can be correctly encoded as a single code point or a sequence of up to 18 code points.[22] An identifier may appear identical but in fact be distinct, such as "Bank of Álpha" and "Bank of Álpha". Normalization[23,24,25] is the process of standardizing string representation such that if two strings are canonically equivalent and are normalized to the same normal form, their byte representations will be the same. Only then can string comparison, ordering and cryptographic operations be relied upon.

In the context of Mina's `client_sdk`, the `string_bits` function implemented on lines of 52-54 of `string_sign.ml` converts an Ocaml string (encoded in UTF-8) into a list of bits for use in the following `derive` function. As a result, the normalization issues described above become relevant. Note that Ocaml support for Unicode has been said to be less than robust. The Mina project does include the latest camomile[26] library in `src/opam.export` but the in-scope code does not appear to use it. The library's OPAM page indicates that it supports Unicode standard 3.2, while the most recent Unicode standard is 14.[27]

Recommendation

The normalization concerns regarding `string_bits` described above and the potential for additional instances across the code base are the primary reason for a Low severity finding (rather than Informational). Perform NFKC[28] Unicode normalization on all strings immediately upon receipt, or constrain them to the set of ASCII values.

Beyond that, the primary purpose of this finding is to raise awareness of several issues related to Unicode encoding. The simplest solution is to constrain (and validate) all strings to the set of ASCII values. If a broader range is instead used, differences in encoding and normalization will need to be handled appropriately.

[19] https://ocsigen.org/js_of_ocaml/3.7.0/api/js_of_ocaml/Js_of_ocaml/Js/index.html#val-to_string
[20] https://github.com/ocsigen/js_of_ocaml/blob/master/runtime/mlBytes.js#L20-L47
[21] http://unicode.org/reports/tr15/tr15-22.html
[22] https://www.compart.com/en/unicode/U+FDFA
[23] https://docs.oracle.com/javase/tutorial/i18n/text/normalizerapi.html
[24] https://blog.golang.org/normalization
[25] https://docs.rs/unicode-normalization/0.1.13/unicode_normalization/
[26] https://opam.ocaml.org/packages/camomile/
[27] https://home.unicode.org/announcing-the-unicode-standard-version-14-0/
[28] See question 2 of https://unicode.org/faq/normalization.html

Finding	**Lack of Domain Separation Tags in Random Oracles**
Risk	**Informational** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-E002674-006
Category	Cryptography
Location	• Multiple instances of Blake2 across the code base (for example) • src/lib/random_oracle/random_oracle.ml
Impact	The lack of domain separation tags in random oracles may impact security proof assumptions and assurances.
Description	There are a number of instances of Blake2 used across the code repository along with the `random_oracle.ml` source file. There does not appear to be any domain separation tags in use, which may impact security proof assumptions and assurances. Cryptographic proofs often rely on the concept of *random oracles* that are instantiated in practice using hash functions. It is considered best practice to include a *domain separation tag* on a per-instance prefix when hashing, so that all uses of oracles in various places of the protocols work on inputs from separate domains. This separation is what allows the different proofs in the protocol to be effectively realized in practice. As this is somewhat outside of the project scope, it was not investigated in depth and is reported out of caution.
Recommendation	Consider the need for domain separation tags.

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

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 mathematical protections for 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.
Timing	Related to race conditions, locking, or order of operations.