Padloc Cryptography Review

Open Technology Fund

August 23, 2019 – Version 1.1

Prepared for

Martin Kleinschrodt

Prepared by

Mason Hemmel

Ava Howell

Synopsis

In Spring of 2019, the Open Technology Fund[1] engaged NCC Group to conduct a cryptographic assessment of the Padloc[2] (formerly known as Padlock) password manager application. This application is a cloud-based password manager that allows for trustless, portable access to passwords for both individual users and organizations. Padloc uses cryptographic constructions to protect the user's vault data, even from the Padloc organization and servers. A Password Authenticated Key Exchange (PAKE)[3] protocol is used to authenticate clients and servers, in an attempt to minimize exposure of the user's password to any passive network attacker or the server. The chosen protocol is Secure Remote Password (SRP).[4]

This assessment focused solely on the cryptographic primitives used by the application and not on the application itself. Source code was provided along with design documentation, and the assessment occurred over two calendar weeks between April 22, 2019 and May 3, 2019.

Scope

NCC Group's evaluated the `ncc-audit` branch at commit `0bf13ce4a9add1d34ab27febbbe4f6be40fa21e9`. Design documentation for the cryptographic protocol was provided through the security whitepaper. As requested, this evaluation was purely focused on cryptographic issues found in the wider code, and explicitly did not address any further application security issues that the application may have had.

The scope of review included the client, server, and shared "core" code, with the exception of any included third-party functionality such as cryptography libraries.

Key Findings

The assessment uncovered a set of cryptographic flaws. Some of the more notable were:

- **Users Removed from Organizations Can Be Silently Re-Added** As discussed in finding NCC-PadlocCryptoReview-013 on page 5, an attacker may replay an organization owner's signature over a user's public key to re-add them to a group from which they had been removed. This could potentially expose groups to credential exfiltration by a user who appeared to have been barred from membership.
- **Authentication Exposes SRP Verifier** As discussed in finding NCC-PadlocCryptoReview-011 on page 7, the unnecessary exposure of an intermediate mathematical value can allow a network attacker to perform a dictionary attack against the user's passphrase. This could potentially allow network attackers to crack passwords for users with passwords of weak to moderate strength.

Strategic Recommendations

- **Consider evaluating OPAQUE** SRP is known to have a relatively weak security proof. One of the properties of SRP, which is explicitly not desired for the security of the Padloc system, is that it is vulnerable to pre-computation attacks (see finding NCC-PadlocCryptoReview-009 on page 19) as well as dictionary attacks in the case where the verifier v is compromised (see finding NCC-PadlocCryptoReview-011 on page 7). There is a newer PAKE design known as OPAQUE,[5] which is not vulnerable to pre-computation attacks and provides a much stronger security proof. OPAQUE represents a solid improvement on the security properties provided by SRP and may warrant consideration for a future iteration of the Padloc authentication scheme.
- **Consider Transitioning Node Cryptography to `node-sodium`** Currently, server-side cryptography is built on Node.js's `crypto` module, which provides an abstraction over OpenSSL's cryptographic hash functions as well as their symmetric and asymmetric cryptographic primitives. However, this library is both lower-level and lower assurance than ideal for applications programming. By contrast, the `node-sodium` library, a Node.js port of libsodium, implements desired higher-level functionality correctly out-of-the-box. As an example, instead of requiring developers to "roll their own" authenticated encryption from a set of primitives, `node-sodium` directly exposes a pair of functions called `crypto_box_easy` and `crypto_box_open_easy` that require minimal further integration or implementation.

[1] https://www.opentech.fund/
[2] https://padloc.app
[3] https://en.wikipedia.org/wiki/Password-authenticated_key_agreement
[4] http://srp.stanford.edu/design.html
[5] https://eprint.iacr.org/2018/163.pdf

Dashboard

Target Metadata

Name	Padloc
Type	Cloud Password Manager
Platforms	Node.js, TypeScript
Environment	Local Instance

Engagement Data

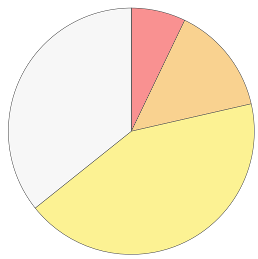
Type	Cryptography Assessment
Method	Code-Assisted
Dates	2019-04-22 to 2019-05-03
Consultants	2
Level of effort	15 person-days

Targets

Source Code https://github.com/padlock/padlock/commits/0bf13ce4a9add1d34ab27febbbe4f6be40fa21e9

Finding Breakdown

Critical Risk issues	0
High Risk issues	1
Medium Risk issues	2
Low Risk issues	6
Informational issues	5
Total issues	**14**



Category Breakdown

Cryptography	9
Data Exposure	1
Timing	3
Other	1

Component Breakdown

Server & Client	6
Server	7
Documentation	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 21.

Title	Status	ID	Risk
Removed Organization Members Can Be Silently Re-Added	Fixed	013	High
Unnecessary Exposure of SRP Verifier v in Protocol Round	Fixed	011	Medium
Member and Organization Signatures Use an Ambiguous Encoding	Fixed	014	Medium
Non-Constant-Time Group Operations	Not Fixed	001	Low
Non-Constant Time HMAC Verification in Server	Fixed	005	Low
Short AES-GCM Tag Length	Fixed	006	Low
Non-Constant-Time Comparison of Session Key Hash `M1`	Fixed	008	Low
Padloc Server Can Execute a Dictionary Attack on Users' Master Passwords	Risk Accepted	010	Low
Non-Constant-Time Comparison over Trusted Device IDs	Fixed	012	Low
Small Order Groups Supported	Fixed	002	Informational
Server Crypto Provider Uses Userland CSPRNG for Cryptographic Randomness	Reported	004	Informational
Missing Check for `a,b > log[g] N` in Key Generation	Not Fixed	007	Informational
SRP is Vulnerable to Pre-Computation Attacks	Risk Accepted	009	Informational
Symmetric Encryption Not Performed According to Specification	Reported	015	Informational

Finding Details

Finding	**Removed Organization Members Can Be Silently Re-Added**
Risk	High Impact: High, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-013
Status	Fixed
Category	Cryptography
Component	Server & Client
Location	`packages/core/src/org.ts`
Impact	By compromising the server and replaying an organization owner's signature, an attacker may be able to silently re-add users who were previously removed from an organization's accessor list, gaining access to the associated vault's password data.

Description

Under Padloc's "Shared-Key Encryption" scheme, multiple vault users can share access to an encrypted vault by providing their public key through the Padloc key exchange protocol. This public key is then signed with the organization owner's private RSA key, authorizing the member's presence inside the organization's accessor list:

```
// packages/core/src/org.ts
// ...snip...
    /**
     * Signs the `member`s public key, id, role and email address so they can be
➜   verified later
     */
    async sign(member: OrgMember): Promise<OrgMember> {
        if (!this.privateKey) {
            throw "Organisation needs to be unlocked first.";
        }

        member.signature = await getProvider().sign(
            this.privateKey,
            concatBytes(
                stringToBytes(member.id),
                stringToBytes(member.email),
                new Uint8Array([member.role]),
                member.publicKey
            ),
            this.signingParams
        );
        return member;
    }
// ...snip...
```

These signatures are then verified by organization members using the owner's public key to provide assurance that the organization owner has authorized the member's access. Once the signature is verified, organization members send the added member the AES key for the vault, encrypted with the member's RSA public key.

The owner's authorizing signature lacks protection against replay attacks. As such, the following attack is possible:

1. Attacker compromises the server. This is required in order to send a new signed accessor list, and to record signatures. Server compromise is specifically permitted within the Padloc threat model.
2. Alice joins an organization, successfully completing the key exchange protocol. The organization owner signs Alice's public key, granting her access.
3. Alice leaves the organization. The organization owner removes her membership details and broadcasts the update to the organization members.
4. Alice's private key is compromised by an attacker, or Alice becomes an attacker.
5. The attacker replays the owner's signature over Alice's public key.
6. Organization members verify the signature provided by the attacker, it validates over Alice's public key.
7. The attacker is granted access to the organization, despite there being no explicit authorization from the organization owner.

This attack is enabled by the lack of replay protections in place on organization owner signatures; thus, past members can always re-add themselves to an organization, even after having been evicted (as long as they control the `members` list). Additionally, if their role changes, they can replay the signature to reset their role to their previous role.

Recommendation Each organization owner signature should contain a unique nonce, with each member of the organization keeping track of the full set of existing nonces. If a client sees a repeated nonce in the organization owner signature, it should consider the signature to be malformed. The nonces should be randomly generated at a size of 4 bytes or longer. It may also be useful for the organization owner to periodically send out signed "reminders" of the current set of burned nonces; if this route is taken, the reminder should be signed and given a timestamp for its period of validity to ensure that attackers cannot replay it.

Retest Results *Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c*

Padloc added a new field called `updated` to the organization member in commit cab32970c3a7c93994a515f6531ddfe66338038c. This field is a monotonically increasing nonce, verified to be greater than or equal to the recorded `minMemberUpdated` property of the `Org` object. When members are removed from an organization, the owner re-signs all members and sets the `minMemberUpdated` property of the organization as well as the `updated` property of each user to the current UNIX time. With this change, attackers can not replay old member signatures to re-add themselves to the organization, since the `updated` field of the replayed member signature would be before `minMemberUpdated` and would be rejected. This will successfully mitigate the vulnerability pointed out in this finding. NCC Group notes that the team did not dynamically validate the fix for this finding due to the time limitations of the retest.

Finding	**Unnecessary Exposure of SRP Verifier v in Protocol Round**
Risk	Medium Impact: Medium, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-011
Status	Fixed
Category	Data Exposure
Component	Server
Location	`packages/core/src/server.ts`, 'packages/core/src/auth.ts
Impact	The password verifier value v is unnecessarily exposed during protocol execution, leading to a significant deviation from the SRP specification. An attacker may perform an active attack to learn verifiers and launch dictionary attacks. If the SRP implementation is executed without TLS, a passive attacker can also execute a dictionary attack on the user's passphrase.

Description

During the server part of round 1 of the SRP protocol, the server sends a Padloc `InitAuthResponse` object to the client that contains the necessary server data called for in round 1 of SRP: the KDF parameters (in this case, salt and iteration count for PBKDF2), and the server's public key ($B = kv + g^b$). In a correct implementation, this exchange does not allow a passive or active attacker to execute a dictionary attack. However, Padloc's `InitAuthResponse` object also contains an `auth` object that carries the `verifier` field:

```
export class Auth extends Serializable implements Storable {
    /** Id of the [[Account]] the authentication data belongs to */
    account: AccountID = "";

    /** Verifier used for SRP session negotiation */
    verifier?: Uint8Array;
```

This violates the security properties of the SRP protocol, since now a theoretical passive attacker has learned the value $v = g^x$, where $x = KDF(p)$, and the attacker can execute a dictionary attack on the user's passphrase with nothing more than this information. An active attacker can connect to the server and initiate the protocol to learn the user's v and execute a dictionary attack. In practice, the active attacker case is made more difficult in Padloc's application due to the fact that the attacker must pass the requirements noted in finding NCC-PadlocCryptoReview-009 on page 19 to initiate the protocol. The passive attack case is mitigated in Padloc's application implementation due to the fact that the protocol is executed over TLS. This situation is analogous to if a traditional password hash based authentication scheme loaded the user's password hash into their login page.

Recommendation

Exclude the verifier field from ever being sent on the wire during execution of the SRP authentication protocol.

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

Padloc successfully stripped the `Auth` object from `InitAuthResponse`, removing the exposed verifier v in commit db5a9f9589e2fe4af39b6bf449c140769e683d53. This successfully remediates the issue.

Finding	**Member and Organization Signatures Use an Ambiguous Encoding**
Risk	Medium Impact: Undetermined, Exploitability: Medium
Identifier	NCC-PadlocCryptoReview-014
Status	Fixed
Category	Cryptography
Component	Server & Client
Location	`packages/core/src/encoding.ts`
Impact	An attacker may be able to produce member objects that validate for other members' signatures, despite having different semantic meaning.

Description

The Padloc protocol uses RSA signatures and HMAC authentication tags for two crucial parts of the system: verifying the identity of organization and member public keys. In both of these cases, the function `concatBytes` is used to encode the relevant data into a `Uint8Array`, which is then signed. The implementation of `concatBytes` lacks a delimiter or any sort of prefix for each of the elements:

```
/**
 * Concatenates a number of Uint8Arrays to a single array
 */
export function concatBytes(...arrs: Uint8Array[]): Uint8Array {
    const length = arrs.reduce((len, arr) => len + arr.length, 0);
    const res = new Uint8Array(length);
    let offset = 0;
    for (const arr of arrs) {
        res.set(arr, offset);
        offset += arr.length;
    }
    return res;
}
```

As such, this encoding is ambiguous. Signatures produced for a given organization member may be valid for other `OrgMember` objects. For example, if the member has the ID "XX", email "test@test.com", role `0x1`, and public key `0x2` (these are dummy values, in reality the public key is an SPKI encoded RSA public key), then the signature for that member will also validate for the member with ID "X", email "Xtest@test.com", role `0x1`, and public key `0x2` since the output of `concatBytes` for both of these objects is identical. In this sense, these signatures are ambiguous. The impact of this finding is undetermined. Due to time constraints NCC Group was unable to determine the full extent of the impact of this property on the security of the Padloc scheme.

Recommendation

Change to a non-ambiguous encoding method for signatures, such as 'TLV', or type-length-value, where the type and length of each data piece is prepended.

Retest Results

Padloc added a delimiter to the encoding scheme in commit fed40f1edd8485805d864ccb038412ef33ca256b, fixing this issue.

Finding	**Non-Constant-Time Group Operations**
Risk	Low Impact: Low, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-001
Status	Not Fixed
Category	Cryptography
Component	Server & Client
Location	`packages/core/src/srp.ts`
Impact	An attacker may be able to learn information about private keys through timing-based side channels.

Description

Padloc includes an implementation of SRP used to authenticate users with the service. The protocol implementation works over a finite field of large prime order $GF(N)$. Operations over this group are performed using the JavaScript `jsbn` 'Big Integer' arbitrary precision arithmetic library:

```
/**
 * Calculates verifier `v` from secret `x` according to the formula
 * ```
 * v = g ^ x % N
 * ```
 */
v(x: BigInteger): BigInteger {
    return this._params.g.modPow(x, this._params.N);
}
```

These `BigIntegers` are not instantiated with a modulus, indicating that their operations cannot be constant-time. In practice, execution time for `jsbn`'s arithmetic was found to be dependent on the inputs. This means that operations that depend on secret keys such as the computation of SRP verifiers ($v = g^x$), ephemeral public keys, and the eventual shared session key, leak information about the secret keys through timing-based side channels.

Recommendation

Consider compiling and utilizing a constant-time modular big integer implementation, such as `BearSSL`'s `i62`.[6] `emscripten`[7] may be used to compile the library to run in a browser environment; for the Node.js environment using native bindings will be sufficient.

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

The code used to compute SRP group operations is unchanged.

[6] https://www.bearssl.org/gitweb/?p=BearSSL;a=tree;f=src/int;h=2fa2ff106f0cf006b83e705855c2f85bf7d76ece;hb=8ef7680081c61b486622f2d983c0d3d21e83caad
[7] https://emscripten.org/

Finding	**Non-Constant Time HMAC Verification in Server**
Risk	Low Impact: High, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-005
Status	Fixed
Category	Timing
Component	Server
Location	`packages/server/src/crypto.ts`
Impact	An attacker-in-the-middle that can precisely measure the timing of the server's HMAC verification may authenticate requests posing as any user.

Description

The server authenticates a client that has an existing account by verifying an HMAC over the session details. The code for this verification is as follows:

```
private async _verifyHMAC(
        key: HMACKey,
        signature: Uint8Array,
        data: Uint8Array,
        params: HMACParams
    ): Promise<boolean> {
        const sig = await this._signHMAC(key, data, params);
        return signature.toString() === sig.toString();
    }
```

The server verifies the HMAC by using its copy of the key to generate the correct HMAC, then uses a string comparison of this known-good HMAC with the one supplied by the user. The comparison function used (===) short-circuits comparison and will immediately stop after hitting the first difference between the two HMACs. An attacker that can make extremely accurate measurements of this process may use the server's verification function as an oracle, creating a statistical model to differentiate between correct and incorrect characters in an HMAC. This allowing the attacker to forge the correct HMAC character-by-character.

The real-world likelihood of this attack is uncertain. First, the attacker would have to create this timing model within the acceptable window of a given timestamp. Second, the machine-level comparison of words tends to occur in 32- or 64-bit words, which deeply limits the effectiveness of this oracle. Nevertheless, it is possible.

Recommendation

Padloc can remove this timing window entirely by comparing cryptographic hashes of the HMACs against each other instead of comparing them directly (i.e. `SHA256(server_hmac) ?= SHA256(client_hmac)`). The collision-resistance property of the cryptographic hash implies that the risk of accepting an incorrect HMAC value is negligibly small, while its preimage-resistance property guarantees that any timing leakage will not leak any information about the underlying HMAC.

Retest Results

Padloc added a function `equalCT` which performs a constant-time comparison and replaced the usage of the default comparison operator in commit 46aea789856569e2e4fa24e171ca85e0a933eb64, successfully remediating this issue.

Finding	**Short AES-GCM Tag Length**
Risk	Low Impact: Low, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-006
Status	Fixed
Category	Cryptography
Component	Server & Client
Location	`packages/core/src/crypto.ts`
Impact	With sufficiently large Padloc vaults, an attacker will have a higher than expected probability of successfully producing a forged vault without knowing the key.

Description

The default authentication tag length for the AES-GCM encryption operations for Padloc vaults is defined as 64 bits:

```
    tagSize: 64 | 96 | 128 = 64;
```

While this tag length is appropriate for small messages, Padloc uses this default tag size to encrypt all of the items in the vault. If the vault grows to a substantial size, the probability of an attacker successfully forging an authentication tag increases,[8] with a forgery succeeding with approximate probability of $n/2^t$ (where `n` is the number of blocks and `t` is the bit size of the authentication tag). Once the attacker successfully creates a forgery, they will learn information about the authentication key, potentially allowing them to forge further messages. NIST guidance recommends a maximum combined ciphertext plus associated authenticated data length of 2^{15} for 64-bit tags.[9] Vaults for large organizations could conceivably be larger than this limit. Successfully creating a forgery requires an active attack: for every forgery that an attacker creates, they must observe whether decryption by the key holder succeeds over the forgery. Such an attack is not practical within Padloc's architecture. While this attack is unlikely to be practical, NCC Group recommends increasing the tag size to 128 bits as a defense-in-depth measure and to conform with NIST guidelines.

Recommendation

Deprecate 64-bit tags and change the default tag size to 128 bits.

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

Padloc changed the default authentication tag size to 128 bits in commit a9c9447068816eb15320cce7075802dbe066b9d9, successfully remediating this issue.

[8] https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf

[9] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

Finding **Non-Constant-Time Comparison of Session Key Hash `M1`**

Risk Low Impact: Medium, Exploitability: Low

Identifier NCC-PadlocCryptoReview-008

Status Fixed

Category Timing

Component Server

Location `packages/core/src/server.ts L148`

Impact An attacker can learn information about the correct value of the authenticator `M = SHA256(A, B, K)`, allowing them to more easily forge the authenticator and add their device as a trusted device, bypassing email-based login verification.

Description Both parties in the SRP PAKE protocol used in Padloc compute a hash `M1` in order to verify that they have reached the same session key, successfully authenticating. This hash is computed as `SHA_256(A, B, K)`, where `A` and `B` are the ephemeral public keys, and `K` is the computed session key. The client computes their `M1` and sends it to the server; the server performs a non-constant-time JavaScript string comparison over the value to authenticate the client.

```
// ...snip... packages/core/src/server.ts
        // Get the pending SRP context for the given account
        const srp = pendingAuths.get(account);

        if (!srp) {
            throw new Err(ErrorCode.INVALID_CREDENTIALS);
        }

        // Apply `A` received from the client to the SRP context. This will
        // compute the common session key and verification value.
        await srp.setA(A);

        // Verify `M`, which is the clients way of proving that they know the
        // accounts master password. This also guarantees that the session key
        // computed by the client and server are identical an can be used for
        // authentication.
        if (bytesToHex(M) !== bytesToHex(srp.M1!)) {
            throw new Err(ErrorCode.INVALID_CREDENTIALS);
        }
```

The JavaScript string comparison operator (`!==`) exits early once the first difference between the two compared strings is detected. This means it has a non-constant-time execution: the execution time of the comparison depends on the contents of the two strings being compared, not just their lengths. If an attacker can construct an accurate statistical model of the timing of this string comparison, they can learn information about the correct `M` value through this timing side-channel. This does *not* directly leak the value of `K`, the session key, due to the security properties of the hash function chosen to compute `M` and `M1` (SHA-256). However, the attacker will be able to pass this authentication check, and add their device as a trusted device:

```
        // Create a new session object
        const session = new Session();
        session.id = await uuid();
        session.account = account;
        session.device = this.device;
        session.key = srp.K!;

    // ...snip...

        // Add device to trusted devices
        const auth = await this.storage.get(Auth, acc.email);
        if (this.device && !auth.trustedDevices.some(({ id }) => id === this.dev
➜   ice!.id)) {
            auth.trustedDevices.push(this.device);
        }
        await this.storage.save(auth);
```

Such an attack may potentially lead to bypass of security mechanisms related to trusted devices. Since the implementation does not remove the `pendingAuth` and demands a new session on authentication failure, the attacker is free to call `createSession` repeatedly to extract timing information about `M`.

Recommendation Change to a constant time comparison, or perform the comparison by taking `SHA256(M) !== SHA256(M1)`.

Retest Results *Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c*

Padloc added a function `equalCT`, which performs a constant-time comparison and replaced the usage of the default comparison operator in commit 46aea789856569e2e4fa24e171ca85e0a933eb64, successfully remediating this issue.

Finding	**Padloc Server Can Execute a Dictionary Attack on Users' Master Passwords**
Risk	Low Impact: Low, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-010
Status	Risk Accepted
Category	Cryptography
Component	Server
Location	`packages/core/src/srp.ts`
Impact	If an attacker compromises the Padloc server, they can execute a dictionary attack against the users' master passphrases using the stored SRP password verifier v.

Description

The Padloc server model is described in the security whitepaper as follows:

> This means that unlike other products, Padloc does not require explicit trust between the end user and the host...Even though v is based on p, it cannot be used to guess the password in case someone eavesdrops on the connection or if the server is compromised. See section 4 of the SRP specification for details.

However, since the Padloc server stores an SRP verifier v, the server can execute a dictionary attack against a user's authentication password. The attack would take the following form:

1. Attacker learns v as well as the key derivation function parameters (stored on the Padloc server).
2. Attacker computes $v' = g^{KDF(p')}$, where the KDF is configured to use the parameters retrieved from the server. p' is the password guess, and v' is the resulting potential verifier.
3. Attacker checks if $v' == v$. If so, they have discovered that $p == p'$, and knows the user's passphrase. If not, repeat from 2.

The cost of each iteration of this dictionary attack is one invocation of the key derivation function with the correct parameters (in this case, PBKDF2), and one modular exponentiation. Given the PBKDF2 parameters used in Padloc, this cost is moderate; however, it could be made much more costly by utilizing a newer password-based key derivation function such as `scrypt`[10] or `argon2`.[11]

Recommendation

This flaw is an unavoidable property of all PAKEs. As a defense-in-depth measure, SRP verifiers v should be treated as sensitive data that may be used to dictionary attack a user's passphrase. For example, verifiers should never be exposed to protocol participants such as in finding NCC-PadlocCryptoReview-011 on page 7. Ensuring that users use high entropy passphrases in addition to using a strong password hashing algorithm is of critical importance here, just as in a traditional authentication context. The documentation should be updated to reflect this property.

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

Since this is an unavoidable property, it is still present. The documentation incorrectly states that v cannot be used to guess the password in case of server compromise. The vulnerability discussed in finding NCC-PadlocCryptoReview-011 on page 7 has been remediated, and there is no longer unnecessary exposure of v.

[10] https://tools.ietf.org/html/rfc7914
[11] https://tools.ietf.org/html/draft-irtf-cfrg-argon2-06

Finding	**Non-Constant-Time Comparison over Trusted Device IDs**
Risk	Low Impact: Low, Exploitability: Low
Identifier	NCC-PadlocCryptoReview-012
Status	Fixed
Category	Timing
Component	Server
Location	`packages/core/src/server.ts`
Impact	An attacker may be able to gain information about the correct device IDs through a timing side-channel, learning about the user's device and gaining the ability to bypass email verification and execute a pre-computation attack.

Description

The Padloc server computes a boolean to determine if a given client is from a trusted device. Devices are given unique identifiers, stored in the server's `auth` construction for that user, and then iterated over when requests are handled as a sort of authenticator:

```
// packages/core/src/server.ts
        const deviceTrusted = auth && this.device && auth.trustedDevices.some(({
➜  id }) => id === this.device!.id);
// ...snip...
        if (!deviceTrusted) {
            if (!verify) {
                throw new Err(ErrorCode.EMAIL_VERIFICATION_REQUIRED);
            } else {
                this._checkEmailVerificationToken(email, verify);
            }
        }
```

Since the JavaScript `===` operator used to verify the request's device ID does not operate in constant time, the execution time of the request handler leaks information about valid device IDs to an unverified, unauthenticated attacker. If the attacker learns a device ID, they can bypass the requirement for email verification of logins, enabling them to initialize a session and learn the user's salt. This is enough to execute a pre-computation attack against the user's passphrase (see finding NCC-PadlocCryptoReview-009 on page 19).

Recommendation

Perform this comparison in constant-time, or calculate `SHA256(id) === SHA256(this.device!.id)`

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

Padloc added a function `equalCT` which performs a constant-time comparison and replaced the usage of the default comparison operator in commit 46aea789856569e2e4fa24e171ca85e0a933eb64, successfully remediating this issue.

Finding **Small Order Groups Supported**

Risk **Informational** Impact: None, Exploitability: None

Identifier NCC-PadlocCryptoReview-002

Status Fixed

Category Cryptography

Component Server & Client

Location `packages/core/src/srp.ts`

Impact In the current design there is no practical impact; however, if in the future it becomes possible for an attacker to select these smaller groups, the security of the SRP authentication scheme will be reduced.

Description The SRP implementation used in Padloc supports the following group sizes:

```
type SRPGroupLength = 1024 | 1536 | 2048 | 3072 | 4096 | 6144 | 8192;
```

SRP reduces to the computational Diffie Hellman assumption. 1024-bit DH keys are known to be insufficient and their discrete log may be partially precomputed by a powerful adversary.[12]

The default group length for SRP clients and servers is 4096. It is not currently possible for an adversary to downgrade to `1024`. However, as a defense-in-depth measure, NCC Group recommends removing all support for lower order groups so there is no possibility of such an attack in the future.

Recommendation Only support larger SRP group sizes, 3072 and above.

Retest Results *Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c*

Padloc removed support for group lengths less than 3072 in commit d4a9a3e12eab517d67b44c161ccf1b032afa4208.

[12] https://weakdh.org/

Finding **Server Crypto Provider Uses Userland CSPRNG for Cryptographic Randomness**

Risk Informational Impact: None, Exploitability: None

Identifier NCC-PadlocCryptoReview-004

Status Reported

Category Cryptography

Component Server

Location `packages/server/src/crypto.ts`

Impact If this randomness is used in future development, Padloc will find itself unnecessarily dependent on users' instances of OpenSSL.

Description Currently, the Padloc server makes use of Node's `crypto.randomBytes()` function. This function makes use of the OpenSSL `RAND_bytes()` function to access the kernel's cryptographically secure pseudo-random number generator (CSPRNG) rather than making direct use of it. While there are not necessarily any issues with the `RAND_bytes()` function, it introduces an unnecessary source of risk while also removing any extra security protections the kernel CSPRNG may provide. For instance, the OpenSSL random number generator is not fork-safe below version 1.1.1[13] or thread safe by default.[14]

Recommendation Consider using a library such as `node-sodium`, which portably leverages kernel-level entropy. This eliminates the unnecessary level of dependence on userland randomness while offering access to well-written cryptographic functions.

[13] For more detail, see https://emboss.github.io/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/.
[14] As per https://wiki.openssl.org/index.php/Random_Numbers#Thread_Safety, the library must be called with `CRYPTO_set_locking_callback`.

Finding **Missing Check for `a,b > log[g] N` in Key Generation**

Risk Informational Impact: None, Exploitability: None

Identifier NCC-PadlocCryptoReview-007

Status Not Fixed

Category Cryptography

Component Server & Client

Location `packages/core/src/srp.ts`

Impact There is no practical impact; however, this represents a divergence from the SRP specification.

Description The SRP protocol calls for the generation of two ephemeral private keys `a,b`, which are integers modulo the order of the protocol group (multiplicative subgroup of the field $GF(N)$). Padloc's implementation of SRP lacks a check that the ephemeral private keys are greater than $\log_g(N)$. This means that there is a probability $< 2^{-t}$, where `t` is defined as the bit-size of the modulus N, that a passive attacker can compute the algebraic logarithm of the public keys `A,B` to discover the ephemeral private keys.

This has no practical implication since the supported modulus sizes are large enough that the probability of such a public key is infinitesimal. However, the check is easy to add, is called for in the SRP specification,[15] and is a good defense-in-depth measure.

Recommendation When generating private keys for SRP, verify that they are greater than $\log_g(N)$.

Retest Results *Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c*

The code for this aspect remains unchanged.

[15] http://srp.stanford.edu/ndss.html#SECTION00044300000000000000

Finding	**SRP is Vulnerable to Pre-Computation Attacks**
Risk	Informational Impact: Medium, Exploitability: None
Identifier	NCC-PadlocCryptoReview-009
Status	Risk Accepted
Category	Cryptography
Component	Server
Location	`packages/core/src/srp.ts`
Impact	If an attacker is able to execute the first round of the SRP protocol with the server, they learn enough information to pre-compute possibly valid passphrases for the targeted user.

Description

Users are authenticated in Padloc using SRPv6, a PAKE protocol. One of the stated design goals of the design is to resist dictionary attacks even if the connection is actively compromised. However, due to the design of SRP, attackers can partially precompute password candidates.

Padloc defines the first round of its authenticated key exchange protocol as follows:

1. Client sends `u`, the username identifier, which in this case is a valid email address, and `A`, the client's public key (g^a).
2. Server sends the salt `s` and KDF parameters, in this case solely the iteration count `i`, to the client for the requested `u`, in addition to the server's public key $B = kv + g^b$.

Since all of the information required to pre-compute candidate passwords for all Padloc users is public, an adversary can execute the following costly attack:

1. Connect to the Padloc server repeatedly, requesting the `s`, `i` parameters for every known user's email addresses.
2. Compute, offline, potential verifiers for common passphrases by taking `x = PBKDF2(s, i, p)` and computing $v_{candidate} = g^x$.
3. Store candidate verifiers in a large lookup database. If the Padloc server is later compromised and password verifiers are revealed, the attacker can leverage their precomputation to quickly crack weak passphrases.

This is mitigated by Padloc's requirement that either the device be trusted, or the user authenticates via email the first stage of the SRP protocol.

Recommendation

Require additional authentication for unauthenticated users to participate in the first round of the protocol. This is currently accomplished though the trusted device and email verification method, which is sufficient.

Retest Results

Retest Performed on July 5, 2019 on commit cab32970c3a7c93994a515f6531ddfe66338038c

Padloc did not change any code relating to this finding.

Finding **Symmetric Encryption Not Performed According to Specification**

Risk Informational Impact: None, Exploitability: None

Identifier NCC-PadlocCryptoReview-015

Status Reported

Category Other

Component Documentation

Location
- security.md#symmetric-encryption
- security.md#simple-symmetric-encryption

Impact Users may be confused by the use of block encryption modes that differ from those discussed in the whitepaper.

Description The Padloc security whitepaper notes multiple times that symmetric encryption only makes use of AES in GCM mode. However, the low-level `_encryptAES` function[16] clearly allows for the use of CCM mode, as shown below:

```
private async _encryptAES(key: AESKey, data: Uint8Array, params: AESEncrypti
➜ onParams): Promise<Uint8Array> {
    if (params.algorithm === "AES-CCM") {
        return SJCLProvider.encrypt(key, data, params);
    }

    const k = await webCrypto.importKey("raw", key, params.algorithm, false,
➜ ["encrypt"]);

    try {
        const buf = await webCrypto.encrypt(
            {
                name: params.algorithm,
                iv: params.iv,
                additionalData: params.additionalData,
                tagLength: params.tagSize
            },
            k,
            data
        );

        return new Uint8Array(buf);
    } catch (e) {
        throw new Err(ErrorCode.ENCRYPTION_FAILED);
    }
}
```

CCM mode does not have any inherent security risks that make it unsuitable, but the implementation does not match the claims of the whitepaper.

Recommendation Padloc should either document the usage of CCM mode in the whitepaper or edit out support in this function.

[16] The relevant code for this snippet can be found at packages/app/src/crypto.ts

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.