Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Audit-Report NIP44 Implementations 11.-12.2023

Cure53, Dr.-Ing. M. Heiderich, Dr. S. Mazaheri, Dr. D. Bleichenbacher

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

“NIP 44 is spec for nostr that aims to add secure encrypted payloads. It can be used to implement end-to-end encrypted direct messaging, among other things.”

From https://github.com/paulmillr/nip44

This report describes the results of a cryptography review of the NIP44 specification, as well as a source code audit against several NIP44 implementations, namely the JS/TS, GoLang and Rust implementations. The project was carried out by Cure53 in late November and early December of 2023.

Registered as *NOS-01*, the examination was requested by Paul Miller in October 2023 and then scheduled to start in November 2023.

In terms of the exact timeline and specific resources allocated to *NOS-01*, Cure53 completed the research as planned, from CW47 to CW48. In order to achieve the expected coverage for this task, a total of ten days were invested. In addition, it should be noted that a team of three senior testers was formed and assigned to preparation, execution, and delivery of this project.

For optimal structuring and tracking of tasks, the examination was split into four separate work packages (WPs):

- **WP1**: Cryptography reviews against NIP44 specification
- **WP2**: Code audits against NIP44 JS/TS implementation
- **WP3**: Code audits against NIP44 GoLang implementation
- **WP4**: Code audits against NIP44 Rust implementation

As the titles of the WPs indicate, white-box methodology was utilized to examine all targets included in the key scope. Cure53 was provided with test-supporting documentation, as well as all further means of access required to complete the tests. Additionally, all sources corresponding to the test-targets were shared to make sure the project can be executed in line with the agreed-upon framework.

Overall, the project progressed effectively. To facilitate a smooth transition into the testing phase, all preparations were completed in CW46, that is in the week preceding the tests. Throughout the engagement, communications were conducted via a private, dedicated and shared Slack channel. Stakeholders - including the Cure53 testers and the members of the relevant development team - could participate in discussions in this space.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The quality of the interactions throughout the test was excellent, with no outstanding queries. These steady exchanges contributed positively to the overall outcomes of this project. The scope was well prepared and clear, which played a major role in avoiding significant roadblocks during the test.

Cure53 offered frequent status updates about the test and the emerging findings. Live-reporting was requested and performed through the aforementioned Slack channel.

The Cure53 team succeeded in achieving very good coverage of the WP1-WP4 targets. All ten findings spotted by the testers were classified as general weaknesses with limited exploitation potential. In other words, no vulnerabilities were detected within the inspected components.

The fact that Cure53 was not able to identify any exploitable vulnerabilities can be interpreted as a positive sign in regard to the security of the NIP44 specification and implementations. Nevertheless, even though the spotted problem can all be seen as general weaknesses and hardening advice, they should not be ignored. It is known that weaknesses may serve as entry points for more severe vulnerabilities in the future. Cure53 strongly recommends swift resolution of all reported flaws.

The following sections first describe the scope and key test parameters, as well as how the WPs were structured and organized. Then, what the Cure53 team did in terms of attack attempts, coverage, and other test-related tasks is explained in a separate chapter on test methodology.

Next, all findings are discussed in the category of miscellaneous flaws, given the absence of exploitable vulnerabilities among the detected problems. Flaws are discussed chronologically. In addition to technical descriptions, PoC and mitigation advice will be provided where applicable.

The report closes with drawing broader conclusions relevant to this November-December 2023 project. Based on the test team's observations and collected evidence, Cure53 elaborates on the general impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the NIP44 specification and the selected implementations in scope.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **Cryptography reviews & code audits against several Paul Miller's NIP44 implementations**
  - **WP1**: Cryptography reviews against NIP44 specification
    - **Repository URL:**
      - https://github.com/paulmillr/nip44/blob/main/spec.md
    - **Commit in scope:**
      - 52648387f41df06ce7cc57a6ba7f75a84883aa46
  - **WP2**: Code audits against NIP44 JS/TS implementation
    - **Repository URL:**
      - https://github.com/paulmillr/nip44/tree/main/javascript
    - **Commit in scope:**
      - 52648387f41df06ce7cc57a6ba7f75a84883aa46
  - **WP3**: Code audits against NIP44 GoLang implementation
    - **Repository URL:**
      - https://github.com/paulmillr/nip44/tree/main/go
    - **Commit in scope:**
      - 52648387f41df06ce7cc57a6ba7f75a84883aa46
  - **WP4**: Code audits against NIP44 Rust implementation
    - **Repository URL:**
      - https://github.com/paulmillr/nip44/tree/main/rust
    - **Commit in scope:**
      - 52648387f41df06ce7cc57a6ba7f75a84883aa46
  - **Additional documentation & information:**
    - https://github.com/nostr-protocol/nips/pull/746
    - https://github.com/nostr-protocol/nips/blob/master/01.md
    - https://github.com/nostr-protocol/nips/blob/master/04.md
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, whilst a vulnerability is present, an exploit may not always be possible. To enable specific follow-up correspondence, each finding has been given a unique identifier (e.g., *NOS-01-001*).

NOS-01-001 WP1: Potential twist attacks in *secp256k1* implementations *(Info)*

Cure53 has evaluated NIP44 in terms of its resilience to twist attacks. Although the current implementation was not found to be vulnerable, further steps can be suggested to make sure this state persists.

Twist attacks are an issue with naive implementations of the *secp256k1* curve. They envelope problems with certain sanity checks, which can cause leakage of sender's private key in the NIP44 protocol.

More precisely, this applies to implementations of *secp256k1*, which accept uncompressed representations of public keys. This means public keys are represented with both *x* and *y* coordinates instead of just *x* and *a* sign-bit. As such, failure to verify whether these public keys are actually on the curve or not should be considered.

Encryption with a key, which is derived from an ECDH-computation on long-term keys of different parties, leaks the private key of the sender. The risk appears when a sender is tricked into encrypting a message with an invalid public key, i.e., one that is not on the real curve, but on a similar curve which admits small subgroups. In this context, the sender can leak their private key. For a more precise description of the twist attack, refer to Lundkvist's blog post[1].

It is noteworthy to mention that the related ECIES encryption scheme, due to not using the sender's private key, does not suffer from this particular issue. Nevertheless, it is still important to avoid such naive implementations, since they may lead to suboptimal security. Similarly, key-exchange variants, where senders provide an ephemeral key pair in addition to their long-term secret key, cannot be securely implemented with uncompressed and unverified public keys.

The current NIP44 implementation includes tests that make sure that deriving keys from invalid public keys fails. Cure53 suggests including more tests specifically designed to catch twist attacks (see NOS-01-003) and also describing valid keys in the specification (see NOS-01-002). It is known that uncompressed and unverified public keys can lead to other security issues.

[1] https://github.com/christianlundkvist/blog/blob/master/2020_[...]_attacks/secp256k1_twist_attacks.md

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NOS-01-002 WP1: Specification improvement recommendations *(Info)*

During the review of the specification, a few opportunities to improve the specification in terms of clarity have been noticed. None of the mentioned points are actually vulnerable, but rather should be viewed as recommended pathways for making implementations easier.

Affected file:
main/spec.md

Recommendations to consider:

- NIP-44 has been designed as a sub-component of NIP-01. This relation should be added to the specification.
- Some parts of the specification directly refer to NIP-01. It would be helpful to identify these parts. For example, the message serialization and signature generation specified in NIP-01 are not mentioned in the current specification of the encryption, but are included in the specification of the decryption.
- The initial counter value for ChaCha20 has not been specified (presumably it is 0). RFC 8439 leaves it open if the initial counter is 0, 1, or anything else. Some primitives - such as ChaCha20-Poly1305 - indeed use 1 as the initial counter.
- It would be helpful if the example given to demonstrate the protocol included some intermediate values, such as the *conversation_key* and the message keys.
- The description should ideally be independent of actual implementations. For instance, the helper method *get_conversation_key* is described as follows:

  Helper method *get_conversation_key*:

  ```
  get_conversation_key(private_key_a, public_key_b):
      secp256k1_ecdh(private_key_a, public_key_b).slice(1, 33)
  ```

  To understand this description, a reader has to understand (or guess) the format of the underlying function *secp256k1_ecdh*.

- The formats of public and private keys are not specified. To avoid confusion, it would be helpful to define the formats. As discussed, the formats used for private key and public key are the same as the ones used for Schnorr signatures (BIP-340). Key validation should include a check that the public key is not 0 and represents a point on the curve.
- For ECDH the two private keys *d* and *n-d* (where *n* is the order of the elliptic curve) are equivalent. Both *d* and *n-d* result in the same public key, since the public key only consists of the *x* coordinate of the scalar multiplication *d*G* and *(n-d)*G*. The *x* coordinate of the ECDH exchange also does not depend on whether *d* and *n-d* are being used for the computation. However, the choice is relevant for Schnorr signatures where the private key is selected among *d* and *n-d,* such that the *y* coordinate of the corresponding public key is an even integer. To avoid confusion, it should be clarified in the specification whether the key generation for NIP44 can

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

generate arbitrary private keys. If so, any joint-use of keys for ECDH and Schnorr signatures needs to include adjustment of private keys when signing.

- The discussion of the properties of the protocol contains the term "*timing leak*". This term may be a bit misleading, since it could be misinterpreted as a protocol flaw that allows determination of secrets through precise timing measurements. It would be clearer to use the term "*metadata information leak*" and an explanation that the protocol does not hide participating parties, location, as well as frequency and date of communication.
- The decryption algorithm includes calculating the HMAC tag and comparing it with the received tag. It is important that this comparison is performed in constant-time, since it otherwise opens a door to timing side-channels, ultimately leading to forgery of HMAC tags. Therefore, the specification should change != comparison and emphasize the usage of a constant-time comparison function for this purpose (see also NOS-01-004).

NOS-01-003 WP1: Proposed test vectors for *invalid_conversation_key* (*Info*)

A very helpful feature of the code is that it contains a good set of test vectors that cover a lot of potential issues. Some other possible flaws mentioned in this document could, ideally, also be converted to test vectors. For example, twist attacks' prevention could entail the following test vectors. These would need to be added to the list pertinent to *invalid_conversation_key*.

Test vectors:

```
  {
       "sec1" :
"0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20",
       "pub2" :
"0000000000000000000000000000000000000000000000000000000000000000",
       "note" : "pub2 is point of order 3 on twist"
  }
  {
       "sec1" :
"0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20",
       "pub2" :
"eb1f7200aecaa86682376fb1c13cd12b732221e774f553b0a0857f88fa20f86d",
       "note" : "pub2 is point of order 13 on twist"
  }
  {
       "sec1" :
"0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20",
       "pub2" :
"709858a4c121e4a84eb59c0ded0261093c71e8ca29efeef21a6161c447bcaf9f",
       "note" : "pub2 is point of order 3319 on twist"
  }
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

It would make sense to add test vectors with truncated cipher-texts to check for out-of-bound access instances. Implementations should throw or return an error instead of going into panic mode. For example, adding the following truncated ciphertexts is advised:

- a ciphertext of size 0 to check for out-of-bound access during version check;
- ciphertexts of size 1..32 to check for out-of-bound access for the salt;
- ciphertexts of size 33..64 to check for out-of-bound access for the MAC.

NOS-01-004 WP2/WP4: Timing differences in HMAC comparison *(Low)*

While reviewing the NIP44 implementations, it was found that the execution of the code leaked some small timing differences. Under the assumption that an attacker can take precise timing measures, key material could possibly be leaked and HMACs forged.

Affected file #1:
noble-ciphers/src/utils.ts

Affected code #1:

```
// Constant-time equality
export function equalBytes(a: Uint8Array, b: Uint8Array): boolean {
  // Should not happen
  if (a.length !== b.length) throw new Error('equalBytes: Different size of
Uint8Arrays');
  let isSame = true;
  for (let i = 0; i < a.length; i++) isSame &&= a[i] === b[i]; // Lets hope
JIT won't optimize away.
  return isSame;
}
```

The function *EqualBytes* was tested with the Compiler Explorer[2]. A slightly simplified version was used with error handling replaced by returning *false*.

Simplified version:

```
// Constant-time equality
export function equalBytes(a: Uint8Array, b: Uint8Array): boolean {
  // Should not happen
  if (a.length !== b.length) return false;
  let isSame = true;
  for (let i = 0; i < a.length; i++) isSame &&= a[i] === b[i]; // Lets hope
JIT won't optimize away.
  return isSame;
}
```

[2] https://godbolt.org/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The result with some added comments is presented below. The code path depends on the value *isSame*, hence the runtime does not represent constant-time.

Results:

```
# Compilation provided by Compiler Explorer at https://godbolt.org/
equalBytes:                             # @equalBytes
        movq    %rdi, -40(%rsp)
        movl    %esi, -32(%rsp)
        movl    %ecx, -16(%rsp)
        movq    %rdx, -24(%rsp)
        movl    -32(%rsp), %eax
        movl    -16(%rsp), %ecx
        movq    -24(%rsp), %rdx
        cmpl    %ecx, %eax
        je      .LBB0_2
        movb    $0, -41(%rsp)
        jmp     .LBB0_8
.LBB0_2:
        movb    $1, -5(%rsp)            # isSame = true
        movl    $0, -4(%rsp)
.LBB0_3:                                # =>This Inner Loop Header: Depth=1
        movl    -4(%rsp), %eax
        movl    -32(%rsp), %ecx
        movq    -40(%rsp), %rdx
        cmpl    %ecx, %eax
        jge     .LBB0_7
        movb    -5(%rsp), %al
        testb   $1, %al                 # Checks whether isSame True or False
        movb    %al, -42(%rsp)          # 1-byte Spill
        jne     .LBB0_5                 # Skips the next lines if isSame is False
        jmp     .LBB0_6
.LBB0_5:                                #   in Loop: Header=BB0_3 Depth=1
        movq    -40(%rsp), %rax
        movslq  -4(%rsp), %rcx
        movb    (%rax,%rcx), %al
        movl    -16(%rsp), %ecx
        movq    -24(%rsp), %rcx
        movl    -4(%rsp), %edx
        movslq  %edx, %rdx
        cmpb    (%rcx,%rdx), %al
        sete    %al
       movb    %al, -42(%rsp)           # 1-byte Spill
.LBB0_6:                                #   in Loop: Header=BB0_3 Depth=1
        movb    -42(%rsp), %al          # 1-byte Reload
        andb    $1, %al
        movb    %al, -5(%rsp)
        movl    -4(%rsp), %eax
        addl    $1, %eax
        movl    %eax, -4(%rsp)
        jmp     .LBB0_3
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
.LBB0_7:
        movb    -5(%rsp), %al
        andb    $1, %al
        movb    %al, -41(%rsp)
.LBB0_8:
        movb    -41(%rsp), %al
        retq
```

The following function *EqualBytes2* is a common way to implement comparison of byte arrays.

***EqualBytes2* function:**

```
function equalBytes2(a: Uint8Array, b: Uint8Array): boolean {
  // Should not happen
  if (a.length !== b.length) return false;
  let diff = 0;
  for (let i = 0; i < a.length; i++) diff |= a[i] ^ b[i]; // This loop is
more difficult to optimize away.
  return diff == 0;
}
```

In this case, the compiler does not generate branches in the inner loop. As such, this part should now be adhering to constant-time.

Results:

```
equalBytes2:                            # @equalBytes2
        movq    %rdi, -40(%rsp)
        movl    %esi, -32(%rsp)
        movl    %ecx, -16(%rsp)
        movq    %rdx, -24(%rsp)
        movl    -32(%rsp), %eax
        movl    -16(%rsp), %ecx
        movq    -24(%rsp), %rdx
        cmpl    %ecx, %eax
        je      .LBB1_2
        movb    $0, -41(%rsp)
        jmp     .LBB1_6
.LBB1_2:
        movl    $0, -8(%rsp)
        movl    $0, -4(%rsp)
.LBB1_3:                                # =>This Inner Loop Header: Depth=1
        movl    -4(%rsp), %eax
        movl    -32(%rsp), %ecx
        movq    -40(%rsp), %rdx
        cmpl    %ecx, %eax
        jge     .LBB1_5
        movl    -8(%rsp), %eax
        movq    -40(%rsp), %rcx
        movslq  -4(%rsp), %rdx
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
        movb    (%rcx,%rdx), %cl
        movl    -16(%rsp), %edx
        movq    -24(%rsp), %rdx
        movl    -4(%rsp), %esi
        movslq  %esi, %rsi
        movb    (%rdx,%rsi), %dl
        movzbl  %cl, %ecx
        movzbl  %dl, %edx
        xorl    %edx, %ecx
        orl     %ecx, %eax
        movl    %eax, -8(%rsp)
        movl    -4(%rsp), %eax
        addl    $1, %eax
        movl    %eax, -4(%rsp)
        jmp     .LBB1_3
.LBB1_5:
        cmpl    $0, -8(%rsp)
        sete    %al
        andb    $1, %al
        movb    %al, -41(%rsp)
.LBB1_6:
        movb    -41(%rsp), %al
        retq
```

Affected file #2:
nip44/go/nip44.go

Affected code #2 (in line 104ff *bytes.Equal* is being used to compare):

```
func Decrypt(conversationKey []byte, ciphertext string) (string, error) {
    [...]
    if !bytes.Equal(hmac_, sha256Hmac(auth, ciphertext_)) {
        return "", errors.New("invalid hmac")
    }
    [...]
}
```

crypto/subtle/ConstantTimeCompare might be used instead.

Affected file #3:
nip44/code/rust/src/lib.rs

Affected code #3 (in line 154ff *!=* is being used for comparison):

```
pub fn decrypt(
       conversation_key: &[u8; 32],
       base64_ciphertext: &str,
       ) -> Result<String, Error> {
       [...]
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
        if mac != calculated_mac_bytes.as_slice() {
        return Err(Error::InvalidMac);
        }

        [...]
}
```

crypto_bigint/macro.const_assert_eq might be used instead.

While these issues would be difficult to exploit, in part due to the fact that NOSTR messages are additionally signed, Cure53 recommends resolving them to be on the safe side.

NOS-01-005 WP3: Missing range checks *(Medium)*

While looking at the code of the Go implementation, Cure53 discovered missing range checks. If the implementation attempts to decrypt an incorrectly formatted ciphertext, it may become possible to cause panic for the code. Generally, there is an expectation that incorrectly formatted data originating from an outside party never causes an implementation to panic, with an error being a more common result in this context.

Affected file:
nip44/go/nip44.go

Affected code:

```
func Decrypt(conversationKey []byte, ciphertext string) (string, error) {
    var ( [...] )
    if ciphertext[0:1] == "#" {
        return "", errors.New("unknown version")
    }
    if decoded, err = base64.StdEncoding.DecodeString(ciphertext); err !=
nil {
        return "", errors.New("invalid base64")
    }
    if version = int(decoded[0]); version != 2 {
        return "", errors.New("unknown version")
    }
    dLen = len(decoded)
    salt, ciphertext_, hmac_ = decoded[1:33], decoded[33:dLen-32],
             decoded[dLen-32:];
   [...]
```

To mitigate this issue, Cure53 recommends adding range checks. Additionally, some test vectors related to this realm are proposed in NOS-01-003.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NOS-01-006 WP1: Lack of forward secrecy *(Medium)*

Forward secrecy is a security property of key-exchange protocols, which guarantees that already exchanged session keys will not be compromised if the static (a.k.a. long-term) secrets of the involved parties are compromised at some point in the future.

Since NIP44 computes the session key via ECDH only on the static keys of the parties, without introducing ephemeral keys, it lacks forward secrecy. In other words, if the static secret key of party *A* is compromised by some adversary, all established session keys that belong to *A* are compromised. As such, the exchanged ciphertexts become decryptable for the given adversary. The NIP44 specification already acknowledges this issue. Notably, right next to key-secrecy and authentication, forward secrecy is widely considered the key goal for any key-exchange protocol.

In an ECDH-based key-exchange protocol, forward secrecy in the context of a compromise of one party is usually achieved by that party introducing additional ephemeral ECDH key material, chosen randomly per session. While introducing such ephemeral key pairs is easy for the sender (a.k.a, key-exchange initiator), incorporating ephemeral keys of the receiver requires either interaction with the sender or pre-sharing ephemeral keys on a public directory, as done by the Signal's X3DH protocol[3].

Forward secrecy on the level of the communication channel, i.e., after the initial session key has been derived, and accounting for compromise of the current session key, it is usually achieved using symmetric ratchets[4]. Pre-shared keys and ratchets introduce considerable complexity to the protocol and may not be a desirable goal for NOSTR at the moment.

As a middle-ground, Cure53 suggests introducing sender-side forward secrecy in the key-exchange, i.e., deriving the session key from both parties' static keys and the sender's ephemeral key.

Assuming static key pairs *($privA_s$, $pubA_s$)* and *($privB_s$, $pubB_s$)*, and ephemeral key pair *($privA_e$, $pubA_e$)*, the session key on the sender's side can be computed by applying an appropriate KDF on ECDH *($privA_s$,$pubB_s$)* and ECDH *($privA_e$, $pubB_s$)*.

[3] https://signal.org/docs/specifications/x3dh/#publishing-keys
[4] https://signal.org/docs/specifications/doubleratchet/#symmetric-key-ratchet

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NOS-01-007 WP1: Lack of key separation in signing and encryption *(Low)*

It was found that NIP44 reused the existing *secp256k1* signature key pairs in the ECDH-based key-exchanges. As a general rule, each key should be only used for one single purpose, so that different algorithms do not interfere with one other.

There are two worries that come to mind when the same key-pair is used for more than one purpose. The first one is that reusing the same key pair for a different purpose is potentially insecure, which can be observed in the case of textbook-RSA, where both schemes break.

However, with elliptic curves there are security proofs for the joint security of EC-Schnorr and ECIES, as well as the joint security of ECDSA and ECIES, where joint security means that the schemes share one key pair[5]. While these proofs do not directly apply to NIP44, the general consensus is that using the same EC key pair for both signing and encryption does not introduce a vulnerability.

The second reason concerns damage control in case of a compromise and the appropriate actions in the face of a compromise detection. If the private key of one scheme is leaked, e.g., due to implementation side-channels, it should ideally not affect the private key used in the other scheme. This is clearly not the case in NIP44.

Introducing a new independent key pair for encryption would complicate the public-key infrastructure, thus indicating that reusing the existing signing key pairs seems to be the more optimal solution for the NOSTR team at the moment.

To achieve key separation to a certain extent without changing the PKI, Cure53 suggests deriving a second private key from the secret signing key using a KDF, computing the corresponding public key, and using this new key pair in ECDH. With this solution implemented, a potentially faulty NIP44 implementation that leaks the private ECDH key would not affect the security of signing. The opposite is, however, not true, as leakage of the signing key would render NIP44 encryption insecure.

[5] https://eprint.iacr.org/2011/615.pdf

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NOS-01-008 WP1: Misuse of the salt parameter in calls to HKDF *(Info)*

It was observed that for each new encryption, a new salt was chosen at random. This salt is used in deriving the encryption and authentication keys from the conversation key. The security definition of KDFs[6], however, explicitly requires that the salt is chosen at random and fixed afterwards. To that end, NIP44 misuses the salt value when calling HKDF, since its usage is not in accordance with the security definition.

More precisely, In *Definition 7* of the HKDF paper[7], the salt *r* is fixed in *Step 2* and the adversary can afterwards query a KDF on different contexts to receive keys of different lengths. Yet, the KDF operates only on the same fixed salt *r* and the same input key material *σ* (Steps 3, 4, and 7). In the rest of the definition, the adversary is challenged to guess whether a specific value comes from the KDF or is a uniformly random value.

Although Cure53 does not believe this issue is exploitable, and the proof of Theorem 1 seems to be adaptable to work with an appropriate variant of *Definition 7*, it is still recommended to comply with the original security definition. Otherwise, NIP44 will not be able to rely on the provable security of HMAC.

It is indeed a good idea to provide a salt value in HKDF. Especially when deploying the Static-Static ECDH, where users re-use their keys in different sessions, salts cater to independence between the derived keys among these otherwise dependent sessions. Furthermore, using different keys and nonces for each encryption makes the protocol more robust against nonce-reuse and randomness failures.

An approach allowing achievement of the security goals whilst complying with the HKDF security definition was discussed with NOSTR. It involves switching the roles of the salt and context information in the input to HKDF. This solution is also used by others[8].

More precisely, the HKDF should be called in each session on the conversation key and the desired output length, together with:

- A fixed salt such as SHA256(*pubA*, *pubB*,"*nip44-v2*"), where *pubA* and *pubB* are the public keys of the involved parties.
- A context information value filled with secure random bytes, chosen freshly for each encryption.

[6] https://eprint.iacr.org/2010/264.pdf
[7] https://eprint.iacr.org/2010/264.pdf
[8] https://soatok.blog/2021/11/17/understanding-hkdf/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NOS-01-009 WP1: No encryption nonce in HMAC tag *(Info)*

While reviewing the encryption procedure, it was observed that the MAC tag was computed only on the ciphertext, without including the nonce used for encryption.

Encrypt-then-MAC constructions of authenticated encryption schemes that require a nonce (a.k.a IV) should authenticate it in the MAC tag[9]. Otherwise, an adversary can modify the nonce, while the tag remains valid for the same ciphertext (although decrypting to a different message).

In other words, not authenticating the nonce in the Encrypt-then-MAC scheme does not produce an authenticated encryption scheme, since the MAC tag is easily forgeable.

While the NIP44 specification suffers from this vulnerability, the overall NOSTR protocol manages to prevent forgeries. The reason is that ciphertexts and MAC tags are transferred together with the salt values, from which nonces are derived, inside a signed NOSTR message. Forgeries are therefore prevented by the outer signature.

Nonetheless, Cure53 suggests modifying NIP44 so that it constitutes a secure authenticated encryption with associated data[10]. This means authenticating the salt value as part of the additional associated data (AAD) in the HMAC tag.

NOS-01-010 WP1: Inclusion of AAD in HMAC with clear boundaries *(Info)*

In the current version of NIP44 only the ciphertext is authenticated via an HMAC tag. This tag does not include any additional associated data (AAD). However, inclusion of the encryption nonce (or the randomness used in encryption, depending on whether and how the protocol is modified) is suggested in NOS-01-008.

Moreover, AAD may become more relevant in future versions of the protocol. Therefore, it is appropriate to also discuss how including additional data should be done to avoid any issues caused by ambiguous parsing. Depending on the nature of AAD, these issues may lead to bugs or security vulnerabilities. A naive solution, which simply concatenates all data, is prone to collisions. Observe that *HMAC (key, a | b | c) = HMAC (key, ab | c)*, where '|' denotes simple concatenation. Even using a special character as a separator does not necessarily solve the issue if the separator character can itself be part of the actual input.

The goal is to enforce clear boundaries between any two inputs of HMAC. For this to happen, it is important to know the number of input items and their lengths. A generic approach is to prepend the input with a 4-byte integer that represents the number of parts being hashed, as well as using a prefix for each of the parts being hashed with a 4-byte integer that represents the length of that part.

[9] https://www.rfc-editor.org/rfc/rfc7366#section-3
[10] https://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

Cure53 concludes that the NIP44 specification and implementations are already quite well-configured and rather successful in preventing some of the most serious attacks. Given that the Cure53 team was unable to spot exploitable vulnerabilities during this *NOS-01* investigation, it can be argued that the project is on the right path in terms of security.

Nevertheless, the findings from this November-December 2023 project in fact include ten general weaknesses. These tickets show that there are still several areas which could benefit from further work, hardening and improvements. Cure53 is certain that after implementing all of the provided recommendations, an excellent level of security could be achieved in the near future by the NIP44 specification, as well as the NIP44 JS/TS, GoLang and Rust implementations.

To offer some clarification on the project, all sources and material were shared prior to the audit and the NOSTR team (Paul Miller) was available during the audit via a dedicated Slack channel. Timely responses to Cure53's questions were incredibly helpful for the testing team. In addition, Cure53 gave updates regarding the audit process, especially zooming in on those issues that required discussion prior to filing.

The goal of NIP44 in its current version 2 is to provide a simple way for the users to communicate privately. Importantly, the lack of certain security guarantees, such as forward secrecy, post-compromise security, deniability, and post-quantum, had been known to the development team prior to *NOS-01*.

Crucially, no exploitable vulnerabilities were found with respect to the current threat model. The protocol is intended to be used together with the basic protocol NIP01. The result of such a use-case is a protocol that first encrypts messages and then signs the ciphertext and context.

When signing a ciphertext, it is important that no party can change the keys, with attention given to the situation of the ciphertext remaining valid but encrypting a different message. Cure53 has looked into this issue and found no weakness. The use of HMAC instead of the more natural POLY1305 prevents such weaknesses.

However, some uncertainty about the protocol persisted, since it is not possible to attribute security guarantees to either NIP01 or NIP44. Instead, one has to look at the interaction between both protocols to argue about security properties. Cure53 found multiple areas that could use hardening and suggested solutions to make sure that the security goals of NIP44 are achieved without overcomplicating the protocol.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The main focus was on the specification, since any security issues or unclear items would/could naturally transfer to the implementations. It is important to have a solid specification that future and more complex versions can be based on. On the positive side, the development of the protocol is done in small and well-defined steps. The developer anticipates future versions and has ensured that offering additional versions is feasible.

More specifically, Cure53 made some editorial suggestions that make the specification more precise and easier to implement (NOS-01-002). Furthermore, it is suggested to improve the protocol by adding a simple form of forward secrecy (NOS-01-006), key separation (NOS-01-007) and authenticated nonces (NOS-01-009). Some modifications in key derivation were suggested to ensure that the protocol can rely on the provable security of HKDF (NOS-01-008).

In the end, the NIP44 specification and implementations appear to have already achieved the security goal of providing users with a way to communicate privately. Miscellaneous issues identified by Cure53 during *NOS-01* correspond to further hardening of the current version and/or suggestions for future versions. Following these suggestions could provide more advanced security guarantees.

In case of deviating from the suggested fixes, Cure53 suggests taking advantage of cryptography-centered consultations beforehand. Scheduling security audits before or after any general modifications is also advised.

Cure53 would like to thank Paul Miller for his excellent project coordination, support and assistance, both before and during this assignment.