Unintended Features of APIs: The OpenSSL Bug That Quietly Disappeared

Nicky Mouha

KeyCryptic

Abstract. This submission to the "Conference for Failed Approaches and Insightful Losses (CFAIL)" revisits Benmocha et al.'s "Unintended Features of APIs: Cryptanalysis of Incremental HMAC" (SAC 2020). The "unintended feature" is that when extra data is added after the HMAC tag has already been computed, most APIs do not raise an error. In the particular implementation used by OpenSSL, it was possible to find key-independent HMAC collisions instantly. GitHub issue #13210 was opened, but no intentional fix was ever implemented. In 2023, #13210 was finally resolved, but only as a side effect of a hardware-token compatibility fix. The fix went unnoticed until 2025, when a regression test finally documented it. This paper argues that the true failure was not the five-year gap in closing #13210, but that the accidental fix resulted in a missed opportunity to gain fundamental insights into the design of cryptographic APIs.

Keywords: CFAIL, OpenSSL, HMAC, Incremental API, Implementation

1 Introduction

In 2020, Caswell opened GitHub issue #13210 [4] with the following question:

> *[Benmocha et al. [2]'s] talk discusses a security attack which may result if the HMAC APIs are used incorrectly. . . . The issue occurs if an application developer makes calls in this order:*
>
> ```
> HMAC_Init_ex()
> HMAC_Update()
> HMAC_Final()
> HMAC_Update()
> HMAC_Final()
> ```
>
> *. . .*
> *As noted above this is incorrect usage of the API, but the APIs do not protect you from this foot gun. Should they?*

The answer should be an obvious "yes." A basic principle in software engineering is: "make it easy to do things right, make it hard to do things wrong." However, it is unclear whether this should be considered incorrect usage, as the documentation neither explicitly permitted nor prohibited it.

The issue is not just theoretical. As Benmocha et al. [2] explained, OpenSSL's HMAC implementation enabled key-independent HMAC collisions when used incrementally. Biham et al. [3] found that another incremental HMAC implementation, used in the Siemens S7 protocol, was less secure than a correct HMAC implementation. This was acknowledged and fixed by Siemens, and assigned CVE-2019-10929 [10].

The OpenSSL issue was labeled `triaged: bug`, and assigned to both the master and 1.1.1 branches. Nevertheless, no efforts to resolve the issue were undertaken for four and a half years.

Since OpenSSL 3.0, the `HMAC_Init_ex`, `HMAC_Update`, and `HMAC_Final` APIs have been deprecated in favor of the `EVP_MAC` interface. EVP is OpenSSL's higher-level API that allows calling different algorithms (in this case, different MACs) without changing too much code. The fact that the HMAC APIs are deprecated did not come up during the discussion and should not diminish the significance of the vulnerability, as any code that is still present, deployed, and reachable can continue to pose a risk.

The vulnerability was only resolved when an unrelated GitHub pull request (PR), intended to support hardware cryptographic tokens and reviewed solely within that context, was merged into the codebase. This change was not supposed to change the default behavior at all. However, an internal guard flag was introduced to the underlying `EVP_Digest` routines. As the HMAC implementation called these routines, it indirectly and inadvertently adopted the new guard, causing the misuse pattern to fail with an error instead of silently producing incorrect tags. Therefore, the default behavior of the HMAC API changed, but this change went unnoticed for another two years.

This paper presents this case study, examines the systemic failures in cryptographic design and implementation, and argues that the accidental fix of Benmocha et al.'s attack resulted in a lost opportunity to understand and improve the design of cryptographic APIs.

2 Analysis of Failures

2.1 Unintended Features of HMAC APIs

The Keyed-Hash Message Authentication Code (HMAC) is defined as follows [7]:

$$\mathrm{HMAC}_K(m) = H\big((K \oplus \mathrm{opad}) \parallel H((K \oplus \mathrm{ipad}) \parallel m)\big),$$

where

- K is the secret key,
- m is the message,
- H is the cryptographic hash function (e.g., SHA-256),
- ipad and opad are the inner and outer padding constants,
- $\oplus$ is the XOR operation, and
- $\parallel$ is the concatenation operation.

For simplicity, it is assumed that the key length is the same as the block length of the underlying hash function. (If this is not the case, HMAC requires that shorter keys are padded, and that longer keys are hashed before they are used.) The applications of H involving "opad" and "ipad" will be referred to as the "outer" and the "inner" hash, respectively.

A straightforward "one-shot" API can be very inefficient. Firstly, protocols such as TLS repeatedly compute intermediate hashes of the handshake data that is transmitted so far, so a one-shot API would require processing the entire transcript from the beginning. Secondly, incremental APIs allow processing the concatenation of several inputs without copying them first into a single contiguous buffer, avoiding costly memory allocation and copy operations.

A typical incremental HMAC API provides the `Init`, `Update`, `Final` operations, similar to implementations of the underlying hash function. It is often assumed that they are called in the correct order: `Init` to initialize data structures, zero or more calls to `Update` to process inputs of an arbitrary length, and `Final` to produce the HMAC tag.

Benmocha et al. [2] classify HMAC implementations into three categories based on what happens when a caller violates this assumption by calling `Update` after `Final`:

Naive. Both inner and outer hash contexts are modified in place. This was used in the incorrect Siemens S7 implementation.

Naive Inner Patched Outer (NIPO). A local copy is made of the outer hash context during finalization, but the inner hash context is modified in place. This corresponds to the vulnerable OpenSSL implementation.

Patched Inner Patched Outer (PIPO). Local copies are made of both the inner and outer hash contexts during finalization. This is a secure construction, as the local copies provide the information to "roll back" `Final` if it is followed by an `Update` call.

A fourth option is missing: Patched Inner Naive Outer (PINO), in which a local copy of the inner hash context is created during finalization. It seems likely that this option was omitted because it has not been observed in real-world implementations.

Benmocha et al. found that the Naive implementation is less secure than a correct HMAC (i.e., either a one-shot or a PIPO implementation). However, the best attacks that they found are of complexity $2^{n/2}$, where n is the output of the hash function or MAC in bits.

It turns out that the NIPO implementation is completely insecure, because it trivially allows the construction of colliding messages. These colliding messages contain parts that are structured as padding (even though they are actually message inputs), but that cannot be distinguished from actual padding (resulting from calls to `Final`).

OpenSSL uses the NIPO implementation, as shown in Listing 1.1. For a Naive implementation, the `EVP_MD_CTX_copy_ex` in Listing 1.1 would be omitted, and the code would continue to operate on the outer context (rather than the inner context, which was overwritten by the outer context).

Listing 1.1. OpenSSL's HMAC NIPO implementation (simplified and annotated) in `crypto/hmac/hmac.c`

```
/* Add len bytes to the running HMAC computation. */
int HMAC_Update(HMAC_CTX *ctx, const unsigned char *data, size_t len)
{
    /* Pass on the input data to the inner hash context. */
    return EVP_DigestUpdate(ctx->md_ctx, data, len);
}

/* Finalize HMAC, store tag in md, length in len. */
int HMAC_Final(HMAC_CTX *ctx, unsigned char *md, unsigned int *len)
{
    /* Will hold length of inner hash value in bytes. */
    unsigned int i;
    /* Will hold inner hash value. */
    unsigned char buf[EVP_MAX_MD_SIZE];

    /* Finalize inner hash, store result in buf, length in i. */
    EVP_DigestFinal_ex(ctx->md_ctx, buf, &i);
    /* Overwrite inner context with copy of outer context. */
    EVP_MD_CTX_copy_ex(ctx->md_ctx, ctx->o_ctx);
    /* Provide inner hash as input to outer hash. */
    EVP_DigestUpdate(ctx->md_ctx, buf, i);
    /* Finalize outer hash, store result in md, length in len. */
    EVP_DigestFinal_ex(ctx->md_ctx, md, len);

    /* This simplified code always returns success. */
    return 1;
}
```

2.2 A Proposed Safe Streaming API

One straightforward approach is to make sure any sequence of API calls is safe.

A typical example of an unsafe API is the C `rand` function: "If `rand` is called before any calls to `srand` have been made, the same sequence shall be generated as when `srand` is first called with a seed value of 1. [6, § 7.20.2.2]" If a developer forgets to call `srand` first, any calls to `rand` result in a well-defined deterministic sequence. This is a frequent source of vulnerabilities (see e.g., [11]). OpenBSD provides an alternative, safe API [12]: calling `srand` before `rand` is not required; in fact, the seed value provided to `srand` is ignored. Here, OpenBSD guarantees that any sequence of calls to `srand` and `rand` is safe, at the cost of intentional non-compliance with the C standard.

For cryptographic applications, Ho et al. [5] proposed a safe, streaming API that can be used for any block-based algorithm.

An error-prone block-based API (Figure 1) requires the caller to navigate a multi-state machine, manually manage internal buffers, and copy the internal state before finalization to extract intermediate results. A safe streaming API (Figure 2) reduces this to a single state, performing buffer management and state copies automatically behind the scenes. If HMAC were implemented through such a safe streaming API, no sequence of calls would be disallowed: `Final` can be called at any time to produce an HMAC tag for the data processed so far, `Update` or `Final` can be called without calling `Init` first (the safe API combines allocation and initialization into one operation), and `Init` can be called at any time to reinitialize the state.

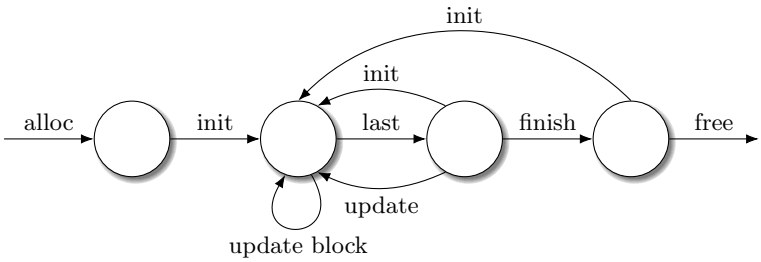

Fig. 1. State machine of an error-prone block-based API (Ho et al. [5])

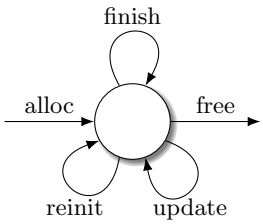

Fig. 2. State machine of a safe streaming API (Ho et al. [5])

This approach is elegant, but it runs into a practical problem when applied to the existing OpenSSL codebase.

A safe streaming API must duplicate the internal hash context so that after calling `HMAC_Final`, the computation can be resumed with `HMAC_Update`. In software implementations, it is relatively straightforward to make a copy of the HMAC context before calling `HMAC_Final`, and use this copy to continue processing data using `HMAC_Update`. This copy can be made inside `HMAC_Final` by a safe API, or by the caller in case the API is possibly unsafe.

However, hardware cryptographic tokens often do not support context duplication. Moreover, in the case of OpenSSL, there is no mechanism to ask whether duplication is supported. The only way to discover this is to try the duplication and observe failure. This tradeoff between API safety and hardware compatibility, further complicated by the lack of capability discovery, will be explored further in Section 2.4.

2.3 The Failure to Resolve the Issue

OpenSSL GitHub issue #13210 [4] was filed in October 2020 with the `triaged: bug` label. In the conversation that followed, there was an agreement that the issue should be fixed, but no fix was ever proposed.

The standard explanations for why security issues remain unfixed (limited resources and other priorities) may apply here, but do not fully account for the

outcome. The HMAC vulnerability was known, acknowledged, and amenable to a simple fix; nevertheless, the issue remained unresolved for four and a half years.

There was, however, some discussion about whether `HMAC_Update` should be allowed after `HMAC_Final`. An argument in favor is that the documentation explicitly allows such behavior for other APIs, e.g., stating that "calls to `EVP_VerifyUpdate` and `EVP_VerifyFinal` can be called later to digest and verify additional data." However, the documentation also states that after calling `EVP_DigestFinal_ex`, no additional calls to `EVP_DigestUpdate` can be made (and this disallowed behavior in the underlying API is exactly what happens when `HMAC_Final` is followed by `HMAC_Update`, see Listing 1.1).

Personal communication with members of the OpenSSL team revealed an additional problem: what would be the "correct" behavior if `HMAC_Update` is allowed after `HMAC_Final`? So far, it was assumed that the HMAC operation should be resumed with additional data provided by `HMAC_Update`. However, the `EVP_DigestFinal` not only computes the digest but also wipes the hash context using `EVP_MD_CTX_reset`. A subsequent `EVP_DigestUpdate` would fail because the context is not initialized. If only changes to `EVP_DigestUpdate` are allowed to fix this, the only option is to perform the missing initialization in `EVP_DigestUpdate`, which would start a new digest computation.

This ambiguity over what the correct Update-after-Final behavior should be (resume or perform a new HMAC computation) led to the conclusion that the safest change was simply to return an error. In software engineering, this is known as the "principle of least surprise." Just returning an error will likely be less dangerous than returning something unexpected, incompatible, or maybe even unsafe.

Lastly, note that the OpenSSL documentation has state transition diagrams for many of its APIs, including the digest API [13]. However, it does not seem that the documentation has such diagrams for `HMAC_Update` and `HMAC_Final`, possibly because this API has been deprecated.

2.4 The Hardware Token Problem

In February 2023, Sorce filed OpenSSL GitHub issue #20364 [14] about a problem that had nothing to do with HMAC. Sorce was working on the pkcs11-provider, which exposes hardware crypto engines through OpenSSL's provider interface. He had discovered that the EVP signing layer attempts context duplication in the `EVP_DigestSignFinal` operation.

Caswell pointed out that the reason for this duplication is explained in the documentation: it is allowed to follow a call to `EVP_DigestSignFinal` by a call to `EVP_DigestSignUpdate` to sign additional data. As mentioned in Section 2.3, other APIs such as `EVP_VerifyFinal` and `EVP_DigestVerifyFinal` allow this as well. For software implementations, `EVP_PKEY_CTX_dup` is a simple memory allocation followed by a copy. But for hardware-backed providers, context duplication may not be possible. As Sorce pointed out [14]:

> *This is something of a big deal for any provider that wants to expose hardware (or remote) crypto engines, like TPM, HSMs, Smartcards, etc.*

A critical subtlety is that there is no mechanism to discover whether context duplication is supported without attempting it. The only way to find out is to call `EVP_PKEY_CTX_dup` and observe whether it returns `NULL`. To understand why, it is necessary to explain how OpenSSL's provider interface works. This explanation is deferred to Section 3.

A preexisting flag, `EVP_MD_CTX_FLAG_FINALISE` (`0x0200`), already allowed applications to opt in to finalization without context duplication. One might ask: Why not simply have the EVP layer automatically set this flag for providers that cannot duplicate? The answer is twofold. First, the flag is an application-level promise not to update the context after finalization; it is an input hint, not a state tracker. Automatically setting it would conflate the application's intent with the provider's capability, and would permanently disable continuation even for providers that could support it. Second, and more fundamentally, the EVP layer cannot know whether a provider can duplicate until it tries. Of course, disallowing `Update` after `Final` altogether is not an option, as this is a documented behavior of OpenSSL for certain APIs that applications are relying on.

Sorce's approach in GitHub PR #20375 [15] addressed both constraints in a single PR. He made context duplication non-fatal: when `EVP_PKEY_CTX_dup` returns `NULL`, the code no longer aborts, but instead falls through and finalizes the original context directly. This single change solved the hardware-token problem: a provider that cannot duplicate is no longer blocked from signing, and the EVP layer does not need to know whether duplication will succeed ahead of time. After the operation completes, the context has been finalized, and any attempt to reuse it should not succeed.

For this, Sorce introduced a new internal flag, `EVP_MD_CTX_FLAG_FINALISED` (`0x0800`), which marks a hashing context as finalized. This flag is set in the signature path when the context duplication was skipped or failed (indicating no additional context can be processed), and unconditionally in `EVP_DigestFinal_ex` (even if the underlying digest operation fails). Once set, any subsequent call to `Update` or `Final` returns an error. In what follows, these two flags are referred to as `FINALISE` and `FINALISED`, respectively.

The `FINALISED` flag thus fulfills two purposes. For the signature path, it guarantees safety after a context was finalized due to a duplication failure, catching any subsequent misuse with a clear error. For the hashing path, it enforces that `EVP_DigestUpdate` must not be called after `EVP_DigestFinal_ex`. As noted in Section 2.3, this pattern was already disallowed by the documentation, so setting the flag after `Final` does not change anything for well-behaved callers of the `EVP_Digest` API.

The rationale given in Sorce's commit message was: "The EVP layer should not rely on the underlying low-level code to handle catching incorrect reuse of contexts." Indeed, the EVP layer needs to be self-sufficient in detecting and preventing incorrect context reuse, independent of provider capabilities. Roeckx objected in the GitHub discussion: "This sounds to me like a recipe for cryptographic failure where you're not sure what you actually did." His concern was that a successful return from `EVP_DigestSignFinal` might suggest the opera-

tion completed normally, when in fact the context was finalized in place because duplication had silently failed. If the caller never makes another `Update` call, the reduced behavior is never discovered.

3 The Provider Interface

To understand why the EVP layer cannot know whether a provider supports `EVP_PKEY_CTX_dup` ahead of time, and indeed, why the very question is more complicated than it appears, it is necessary to examine the architectural context in which Sorce's PR was situated: OpenSSL's provider interface. A brief description is provided here; see Belyavskiy [1] for more details.

3.1 What Providers Are

Introduced in OpenSSL 3.0, providers are the mechanism by which OpenSSL loads algorithm implementations. A provider is a unit of code, built into the OpenSSL library itself or loaded as a shared object, that registers a set of algorithm implementations. The default provider supplies standard algorithms (AES, SHA-256, RSA, etc.).

The provider system replaced the earlier Engine API, which had been criticized for requiring applications to explicitly configure alternative implementations. Providers were designed to be transparent: an application should not need to know whether the SHA-256 it calls is provided by the default provider or by a hardware token.

The transparency goal led to a design where the set of available algorithms is mutable at runtime. Providers can be loaded and unloaded during program execution.

3.2 The Connection to Context Duplication

The provider interface explains why Sorce's "try and see" approach to context duplication is necessary, and why any attempt to statically determine provider capabilities would be fragile.

The set of providers, and therefore the set of available algorithm implementations, is not fixed at compile time or even at process start. It can change as providers are loaded and unloaded. Moreover, the provider interface does not define a capability query for "does this algorithm implementation support `EVP_PKEY_CTX_dup`?" The only possible approach here is to try and see.

This is a consequence of the same design philosophy that made the provider interface generally difficult. The flexibility to load and unload algorithms dynamically is paid for in complexity throughout the stack, including in seemingly unrelated areas like context duplication. Sorce's PR is a case study of this larger tradeoff: the general solution (non-fatal context duplication failure with a guard flag) is more complex than the simpler alternative (disallow `Update` after `Final`), but it preserves the flexibility that the provider interface was designed to enable.

4 The Accidental Fix

Sorce's PR change was reviewed solely in the context of hardware-token compatibility. No reviewer mentioned HMAC. But when `FINALISED` was set unconditionally in `EVP_DigestFinal_ex`, this meant that every usage of that function (including in the HMAC implementation) inherited the guard. The HMAC code path calls `EVP_DigestFinal_ex` to finalize the outer hash; after the PR, this call sets `FINALISED` on the working context. A subsequent `HMAC_Update`, which delegates to `EVP_DigestUpdate`, would then hit the new guard and return an error instead of silently producing an incorrect tag. Issue #13210 was closed, not by an intentional HMAC fix, but as an unnoticed side effect.

The HMAC fix was thus an accident: it arose from a change whose primary purpose (handling context duplication failure gracefully) was entirely unrelated to the one that actually fixed the HMAC vulnerability. The `FINALISED` flag was introduced to protect against the reuse of contexts that had been finalized in place due to duplication failure, but its unconditional application in `EVP_DigestFinal_ex` made it a general-purpose guard against Update-after-Final for all constructions that use a hash function, including HMAC.

To resolve issue #13210, the regression test `test_hmac_final_update_fail` was added in [8] to confirm that an error is returned when `HMAC_Final` is followed by either `HMAC_Update` or `HMAC_Final`. While preparing this submission to CFAIL, it was noticed that there is no mention of this in the HMAC documentation, so a PR was submitted to address this [9].

5 The Lost Lessons

Documentation is often incomplete. The OpenSSL documentation stated that `EVP_DigestUpdate` must not be called after `EVP_DigestFinal_ex`. But calling `EVP_DigestSignUpdate` after `EVP_DigestSignFinal` to process more data is explicitly allowed. The HMAC documentation did not have any statements about `HMAC_Update` after `HMAC_Final`. And even if it did, documentation alone is not enough to enforce any behavior.

API ossification constrains the solution space. It is not possible to simply disallow `Update` after `Final` everywhere, because it has always been allowed to call `EVP_DigestSignUpdate` after `EVP_DigestSignFinal`, and applications rely on this behavior. Once an API is widely deployed, its behavior ossifies, so a "creative" solution was necessary to support certain hardware tokens without breaking existing applications.

The impossibility of capability discovery. Before PR #20375, there was no way for the EVP layer to handle a provider that could not duplicate its context. The only way to discover incompatibility was to try, and trying resulted in a hard failure. This is a general lesson for cryptographic API design: when an operation may fail due to provider-specific constraints, the API must be designed

to handle that failure gracefully rather than treating it as unrecoverable. The PR's solution (make the failure non-fatal and guard against the consequences) is one approach, but it leaves the caller uncertain about whether the operation completed normally or in a degraded mode (as Roeckx noted).

Accidental fixes are invisible. The HMAC vulnerability was resolved not by an intentional fix, but by a guard flag introduced for an unrelated purpose. Because no one was looking for the fix, no one noticed it had happened. The change went undetected for two years, and when it was discovered, it was through a regression test update, not through any security or audit process. An accidental fix cannot be reviewed, tested, or depended upon. It does not appear in release notes, changelogs, or security advisories. Users of older OpenSSL versions have no way to know that upgrading would close this vulnerability.

The lost opportunity. The most instructive aspect of this story is not that the vulnerability existed, but that its eventual resolution produced no learning. The PR's changes were reviewed in the context of hardware-token compatibility. The HMAC oversight was a separate failure (in API design, in issue triage, in code review scope) that was corrected by accident rather than by design. The lessons that could have been learned from a deliberate HMAC fix (about state-machine enforcement, about documentation versus enforcement, about the design of safe streaming APIs) were never learned.

6 Conclusion

The OpenSSL HMAC misuse vulnerability was known, documented, exploitable, and fixable for four and a half years. It was resolved by accident, through a change whose purpose was entirely unrelated, and the resolution went unnoticed for two more years. The sequence of events shows failures at every level of the software security pipeline: API design that prioritizes convention over enforcement, issue triage that cannot prioritize acknowledged vulnerabilities, code review that examines a patch only within its stated scope, and institutional processes that lose the lessons of accidental fixes.

The underlying technical story is instructive in its own right. The PR that accidentally fixed HMAC was addressing a fundamental constraint in the provider interface (Section 3): there is no way to discover whether a provider supports context duplication without attempting it, and the only robust approach is to handle failure gracefully. The resulting design (non-fatal duplication failure paired with a finalized-context guard) is a clean solution to a hard problem. But because the HMAC fix was a side effect, the connection was never made between the two issues. The broader lesson is that architectural decisions, such as the provider interface's commitment to dynamic algorithm replacement, ripple into seemingly unrelated areas like HMAC security, with consequences that are difficult to anticipate and harder to detect.

This paper is submitted to CFAIL in the hope that this case study of a failure cascade will offer the community valuable insight into the continual challenges

involved in maintaining widely deployed software. Furthermore, hopefully, the lessons drawn from this experience will contribute to the improvement of cryptographic API design.

References

1. Belyavskiy, D.: OpenSSL in Red Hat Enterprise Linux 10: From engines to providers (Nov 2024), https://www.redhat.com/en/blog/openssl-3-providers-rhel-10
2. Benmocha, G., Biham, E., Perle, S.: Unintended Features of APIs: Cryptanalysis of Incremental HMAC. In: Dunkelman, O., Jacobson, M.J.J., O'Flynn, C. (eds.) Selected Areas in Cryptography - SAC 2020 - 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers. pp. 301–325. Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-81652-0_12
3. Biham, E., Bitan, S., Carmel, A., Dankner, A., Malin, U., Wool, A.: Rogue7: Rogue Engineering-Station attacks on S7 Simatic PLCs. Black Hat USA 2019 (Aug 2019), https://i.blackhat.com/USA-19/Thursday/us-19-Bitan-Rogue7-Rogue-Engineering-Station-Attacks-On-S7-Simatic-PLCs-wp.pdf
4. Caswell, M.: Incorrect usage of the HMAC APIs. OpenSSL issue #13210 (Oct 2020), https://github.com/openssl/openssl/issues/13210
5. Ho, S., Fromherz, A., Protzenko, J.: Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification. Proc. ACM Program. Lang. **7**(ICFP), 385–416 (2023). https://doi.org/10.1145/3607844
6. ISO/IEC: ISO/IEC 9899:1999 – Programming Languages – C (1999), https://www.iso.org/standard/29237.html
7. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Feb 1997). https://doi.org/10.17487/RFC2104
8. Mouha, N.: Regression test for incorrect HMAC API usage. OpenSSL pull request #27692 (May 2025), https://github.com/openssl/openssl/pull/27692
9. Mouha, N.: doc: document that HMAC_Update cannot be called after HMAC_Final. OpenSSL pull request #31570 (Jun 2026), https://github.com/openssl/openssl/pull/31570
10. National Vulnerability Database: CVE-2019-10929 (Jun 2026), https://nvd.nist.gov/vuln/detail/CVE-2019-10929
11. National Vulnerability Database: CVE-2019-11690 (Jun 2026), https://nvd.nist.gov/vuln/detail/CVE-2019-11690
12. OpenBSD Manual Pages: rand(3) — bad pseudo-random number generator (Dec 2014), https://man.openbsd.org/OpenBSD-6.0/rand.3
13. OpenSSL Documentation: life_cycle-digest (Jun 2021), https://docs.openssl.org/3.1/man7/life_cycle-digest/
14. Sorce, S.: Context duplication is often unsupportable with hardware tokens. OpenSSL issue #20364 (Feb 2023), https://github.com/openssl/openssl/issues/20364
15. Sorce, S.: Do not fail if ctx dup does not succeed. OpenSSL pull request #20375 (Feb 2023), https://github.com/openssl/openssl/pull/20375