Sweet B

Security Assessment

January 24, 2020

Prepared For:
Brian Mastenbrook | *Western Digital*
Brian.Mastenbrook@wdc.com

Prepared By:
Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Paul Kehrer | *Trail of Bits*
paul.kehrer@trailofbits.com

Executive Summary

From January 13 through January 24, 2020, Trail of Bits reviewed the security of Sweet B, a library that provides elliptic curve operations over 256-bit prime fields and a set of supporting hash-based primitives. Trail of Bits conducted this assessment over the course of four person-weeks with three engineers working from commit `02d41f4d` of `sweet-b`.

During the first week, we verified we could build and run tests for the codebase, then evaluated the output for several static analyzers on the code. We identified functions expected to have constant-time behavior for further testing and manually reviewed `SHA256`, `HMAC_SHA256`, `HMAC_DRBG`, and `HKDF` for security and compliance with relevant standards.

In discussions with Western Digital, we identified an opportunity to provide empirical evidence that Sweet B maintains two important security properties: that certain functions maintain constant-time behavior and that the library works as expected in many build scenarios.

During the second week, we used a modified version of QEMU to obtain instruction traces for functions where constant runtimes were a concern. We verified that the instruction traces did not vary with the input to the function. We also checked those instruction traces for certain problematic instructions.

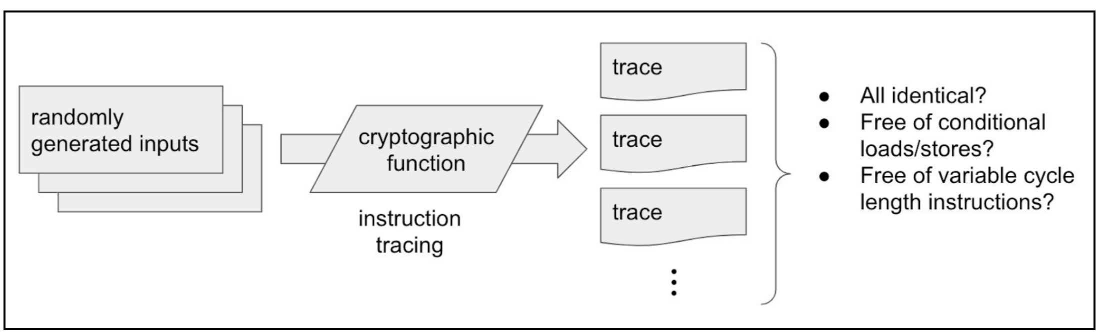

Figure 1: We tested Sweet B in a manner similar to fuzzing to identify possible timing issues

Instruction trace analysis identified a potential misconfiguration that could produce functions that are not constant time (TOB-SB-001) and that undue trust was placed in the behavior of certain `libc` functions (TOB-SB-003). See Appendix C for more details.

We performed a differential analysis of possible build configurations to ensure the compiled results did not produce unexpected or broken behavior. Compiler output and unit tests provided the basis for evaluation of different builds.

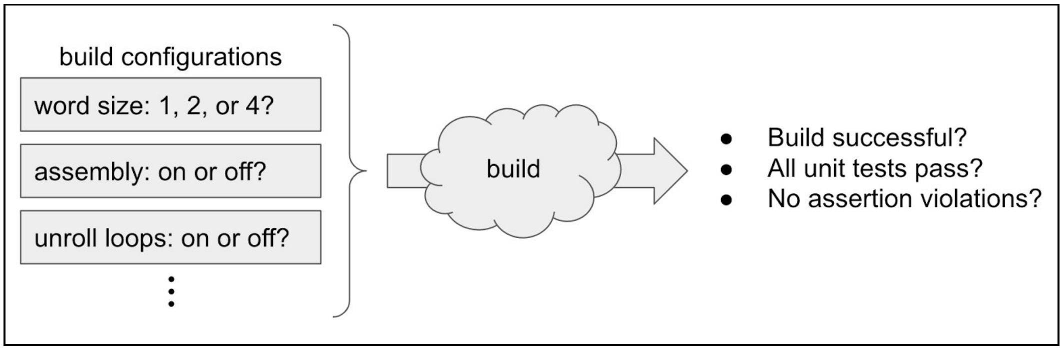

Figure 2: We performed differential testing of possible build configurations

Build configuration analysis identified that the library could produce incorrect results and possibly read memory out-of-bounds due to certain assembly instructions (TOB-SB-004). See Appendix E for more details.

We also more closely reviewed the unit tests provided with Sweet B. We performed code coverage analysis of the unit tests to identify possible gaps in those tests, then recommended areas for improvement. See Appendix D for more details.

Finally, we completed manual review of the elliptic curve and prime-field implementations. We identified potentially error-prone functions (TOB-SB-002) and issues related to the ECDSA API (TOB-SB-005). We found no issues regarding standards compliance.

Throughout our review, the quality and abundance of the comments within the code significantly aided our diagnoses of the issues we found.

Project Dashboard

Application Summary

Name	Sweet B
Version	`02d41f4d`
Type	C, Thumb assembly
Platforms	ARM

Engagement Summary

Dates	January 13 through 24, 2020
Method	Whitebox
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	2	■■
Total	6	

Category Breakdown

Configuration	1	■
Cryptography	2	■■
Data Validation	1	■
Timing	2	■■
Total	6	

Engagement Goals

The engagement was scoped to provide a security assessment of standards compliance, constant time behavior, unit testing, build configuration, and safety/usability of the API.

Specifically, we sought to answer the following questions:

- Do the functions properly implement their respective standards?
- Do the constant time functions produce identical instruction traces when presented with distinct inputs?
- Are there gaps in the unit tests?
- Do the unit tests pass under all possible build configurations?
- Are there aspects of the API that seem error-prone or unintuitive?

Coverage and Compliance

This section discusses our manual coverage of the Sweet B codebase. Specifically, we describe which components we analyzed to determine their compliance with their corresponding specifications. We also comment on test vectors and unit tests supplied for the corresponding primitives.

SHA256: Trail of Bits reviewed the code corresponding to Sweet B's `SHA256` implementation and its corresponding test vectors. Test vectors were supplied from both FIPS 180-2 and the NIST cryptographic algorithm validation program (CAVP). These vectors are designed to exercise potential edge cases in the algorithm and provide some assurance of implementation correctness. The Sweet B code passes all of these test cases. The implementation was also assessed for its compliance with the NIST FIPS 180-4 standard. Specifically, this review determined if the implementation's parameters (e.g., word size) and general interface comply with the NIST standard.

HMAC_SHA256: Trail of Bits reviewed the code and corresponding test vectors for Sweet B's `HMAC_SHA256` implementation. The implementation passes all test cases, with test vectors supplied from Internet Engineering Task Force (IETF) RFC 4231. This implementation's parameters and general interface were also assessed for their compliance with NIST FIPS 198-1.

HMAC_DRBG: Trail of Bits additionally reviewed the code and test vectors for Sweet B's `HMAC_DRBG` implementation. The implementation passes all tests, with test vectors supplied from the NIST CAVP. Our review ensured that parameters fixed by the implementation and parameters adjustable by users only take values that are in accordance with NIST SP 800-90 standards. Further, this review determined whether the implementation interface and error-handling comply with the standard. It has been shown that `HMAC_DRBG`, even when compliant with NIST SP 800-90, is not backtracking-resistant when additional input is not required upon generating random bits. See TOB-SB-006 for more details.

HKDF: Trail of Bits also reviewed the code and test vectors for Sweet B's `HKDF` implementation. This implementation was assessed for its compliance with NIST SP 800-108 and IETF RFC 5869. The test vectors were obtained from IETF RFC 5869, and the Sweet B code passes of all these test cases. This review ensured that both the parameters and interface complied with the appropriate standards. In particular, this review verified that the interface handles special inputs correctly, such as a `NULL` salt.

P-256, secp256k1: Trail of Bits reviewed the instantiation of two elliptic curves, `P-256` and `secp256k1`, used throughout the Sweet B codebase. `P-256` was assessed for its compliance with NIST FIPS 186-4, and `secp256k1` was assessed for its compliance with the Standards for Efficient Cryptography (SECG) SEC 2. This review ensured that the constants and parameters for each curve were specified correctly.

Prime field and EC arithmetic: Trail of Bits reviewed the code implementing prime-field and elliptic curve arithmetic for its compliance with the Handbook of Applied Cryptography and the paper Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. Our assessment covered:

- General correctness of algorithms
- Sanity-checking of functions' constant time behavior
- Proper handling of errors and special cases
- Proper side-channel mitigation with blinding

In general, unit testing of the arithmetic was strong. Negative testing for failure conditions and special case values were handled (see Appendix D for more detail).

ECDH: Trail of Bits reviewed the code and test vectors for Sweet B's `ECDH` implementation. This implementation was assessed for its compliance with NIST SP 800-56A Revision 3. Test vectors were supplied from NIST CAVP for the `P-256` curve, and separate testing was supplied for the secp256k1. Randomized testing was also observed for both curves. This review ensured that parameters and error-handling were compliant with the standard.

ECDSA: Trail of Bits reviewed the code and test vectors for Sweet B's `ECDSA` implementation. This implementation was assessed for its compliance with NIST FIPS 186-4. As with `ECDH`, test vectors from NIST CAVP were supplied for `P-256`, and separate unit testing was supplied for secp256k1, with randomized testing additionally supplied for both. This review ensured that the parameters and error-handling were compliant with the standard. Particular care was given to ensure nonce generation was occurring safely. Sweet B provides two modes of nonce generation: deterministic generation using RFC 6979, and randomized generation using `HMAC_DRBG`. In the case of `HMAC_DRBG`, both the private key and message are used as additional input to ensure sufficient entropy. Test vectors from RFC 6979 were also supplied.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Document the fact that `SB_DEBUG_ASSERTS` should not be enabled in production code** (TOB-SB-001). If enabled, these asserts could violate constant time behavior.

❑ **Consider replacing SDL banned functions with their recommended alternatives** (TOB-SB-002). Western Digital noted that the recommended alternative functions in Table 2.1 are not available in the version of `libc` that they link against. Compiling those functions individually should be considered (see also TOB-SB-003). If this is not an option, we recommend switching to those functions if and when they become available.

❑ **Provide your own implementation of any `libc` function required by a function that must be constant time. Use `-ffreestanding` when compiling and verify that compiled code is not affected by GCC bug 56888** (TOB-SB-003). Otherwise, constant time guarantees cannot be provided.

❑ **Add compile time checks to ensure that `SB_WORD_SIZE` is 4 and `SB_FE_VERIFY_QR` is off when `SB_FE_ASM` is enabled, and add a static assertion to verify that the offset of `p_mp` assumed by the assembly code is correct.** (TOB-SB-004). This will prevent build configurations that are incompatible with the assembly code.

❑ **Update the comments surrounding `sb_sw_sign_message_digest` and similar functions to better inform users about securely digesting messages. Specifically, alert users that it is imperative that `sb_sha256` is used for digesting messages** (TOB-SB-005). This will help ensure that users digest their messages securely.

❑ **Add higher-level routines to the API that provide safer alternatives** (TOB-SB-005)**.** Specifically, add a function `sb_sha256` that combines `sb_sha256_init`, `sb_sha256_update`, and `sb_sha256_finish.` Also, add a function `sb_sw_sign_message` that would take as input a raw message, securely digest it using `sb_sha256`, and then pass the secure digest as input into the `sb_sw_sign_message_digest` function. These functions would serve as guardrails for users looking to sign messages.

❑ **Document the limitations of `HMAC_DRBG` for users** (TOB-SB-006)**.** Users may not be aware that backtracking resistance of `HMAC_DRBG` is predicated upon additional input.

❑ **Add unit tests to achieve complete coverage of Sweet B.** Additional coverage by unit tests will help ensure the library remains functional as development continues, and enhances the efficacy of the build system analysis (Appendix D).

Long Term

❑ **Produce a guide for developers on how to incorporate Sweet B into their projects** (TOB-SB-001). The guide should list which configurable defines are safe to enable in production code. The guide could also mention the types of errors that Sweet B checks for generally (e.g., whether a point is on a curve) and the types of errors that it ignores (e.g., whether a required argument is null). Such a guide would make it less likely for a developer to use Sweet B incorrectly.

❑ **Add a build target composed of the object files of all constant time functions, and verify that the target has no external dependencies** (TOB-SB-003). This can give further assurance of constant time guarantees.

❑ **Consider whether a solution that does not use a hardcoded offset within the assembly code would be preferable** (TOB-SB-004). Such a solution would help avoid further issues with build configurations.

❑ **As additional digest functions are added to Sweet B (e.g., SHA-3, SHAKE256), add unit tests to verify their compatibility with `sb_sw_sign_message_digest`** (TOB-SB-005). This will help to ensure the correctness of `sb_sw_sign_message_digest`. Such unit tests could also serve as examples of `sb_sw_sign_message_digest`'s proper use.

❑ **Consider methods to promote the use of additional data in `HMAC_DRBG`, or switch away from it** (TOB-SB-006)**.** Refactor the code to make use of additional data easier and avoiding use of additional data more difficult. Consider replacing `HMAC_DRBG` with `Hash_DRBG`.

Findings Summary

#	Title	Type	Severity
1	Assembly does not work in all build configurations	Configuration	Medium
2	Use of libc functions that may not be constant time	Timing	Low
3	Enabling of SB_DEBUG_ASSERTS violates constant time behavior	Timing	Low
4	HMAC_DRBG may lack backtracking resistance	Cryptography	Low
5	Use of functions on the SDL List of Banned Functions	Data Validation	Informational
6	API for ECDSA signatures does not enforce secure message digests	Cryptography	Informational

1. Assembly does not work in all build configurations

Severity: Medium
Type: Configuration
Target: `sb_fe_armv7.s`

Difficulty: High
Finding ID: TOB-SB-004

Description

The assembly implementation of `sb_fe_mont_mult` makes assumptions about the build configuration that are not guaranteed.

In particular, the assembly implementation assumes that `SB_WORD_SIZE` is 4. As can be seen in Figure 4.1, the code operates on words of size 4.

```
sb_fe_mont_mult:
   push {r4, r5, r6, r7, r8, r9, r10, r11, lr}


   ldr ip, [r3, #32] /* use ip as p->mp */
...
.set sb_i, 0
.rept 8 /* for (i = 0; i < 32; i += 4) */
...
   ldr r8, [r1, #sb_i] /* use r8 as x_i */
...
.set sb_i, sb_i + 4
.endr
```

Figure 4.1: `sb_fe_armv7.s#L442-L528.`

Similarly, the assembly implementation assumes that the offset of the field `p_mp` within `sb_prime_field_t` is 32 (see the line beginning with `ldr` in Figure 4.1). However, this is likely to hold only if `SB_FE_VERIFY_QR` is off (see Figures 4.2 and 4.3), and even then it is not guaranteed.

```
typedef struct sb_prime_field_t {
   /** The prime as a \ref sb_fe_t value. */
   sb_fe_t p;


   /** -(p^-1) mod M, where M is the size of \ref sb_word_t . */
   sb_word_t p_mp;
```

```
    ...
} sb_prime_field_t;
```

Figure 4.2: `sb_fe.h#L387-L392.`

```
typedef struct sb_fe_t {
    sb_word_t words[SB_FE_WORDS];
#if defined(SB_FE_VERIFY_QR) && SB_FE_VERIFY_QR != 0
    _Bool qr, qr_always;
    const struct sb_prime_field_t* p;
#endif
} sb_fe_t;
```

Figure 4.3: `sb_types.h#L169-L175.`

Exploit Scenario

Alice develops an embedded device and chooses Sweet B to provide its cryptographic capabilities. Alice leaves `SB_FE_VERIFY_QR` enabled in her production code. The incorrect field offset causes her code to fail in ways that reveal the contents of sensitive memory. Eve exploits this fact to steal Alice's clients' cryptographic material. Alice is forced to perform an expensive software update and/or product recall.

Recommendation

Short term:

1. Add compile time checks to ensure that `SB_WORD_SIZE` is 4 and `SB_FE_VERIFY_QR` is off when `SB_FE_ASM` is enabled.
2. Add a static assertion to verify that the offset of `p_mp` assumed by the assembly code is correct.

Long term, consider whether a solution that does not use a hardcoded offset within the assembly code would be preferable.

2. Use of libc functions that may not be constant time

Severity: Low
Type: Timing
Target: `sb_sw_lib.c`

Difficulty: High
Finding ID: TOB-SB-003

Description

Some functions within `sb_sw_lib.c` that are required to be constant time call `libc` functions. However, if the `libc` function makes no guarantees about the time it takes, the constant behavior of the caller may be violated.

An example appears in Figure 3.1. The function `sb_sw_compress_public_key` is supposed to be constant time. However, `sb_sw_compress_public_key` calls `memcpy`, which provides no guarantees about the amount of time it takes.

```
sb_error_t sb_sw_compress_public_key(sb_sw_context_t ctx[static const 1],
                                     sb_sw_compressed_t compressed[static const 1],
                                     _Bool sign[static const 1],
                                     const sb_sw_public_t public[static const 1],
                                     sb_sw_curve_id_t curve,
                                     sb_data_endian_t e)
{
    ...
    // Copy the X value to the compressed output.
    memcpy(compressed->bytes, public->bytes, SB_ELEM_BYTES);
    ...
}
```

Figure 3.1: `sb_sw_lib.c#L1617-L1651.`

Exploit Scenario

Bob produces his own version of `libc`. As a precaution against memory corruption, Bob's `memcpy` performs the copy as normal, but then performs a second pass to verify that the number of set bits in the source and destination match. Alice compiles Sweet B and links against Bob's `libc`. Bob's version of `memcpy` introduces a timing side-channel that allows Eve to steal Alice's cryptographic material.

Recommendation

Short term, provide your own implementation of any `libc` function required by a function that must be constant time. Use `-ffreestanding` when compiling and verify that compiled code is not affected by GCC bug 56888. Finally, consider using the recommended alternatives in Table 2.1 (e.g., using `memcpy_s` in place of `memcpy`).

Long term, add a build target composed of the object files of all constant time functions, and verify that the target has no external dependencies.

3. Enabling of SB_DEBUG_ASSERTS violates constant time behavior

Severity: Low
Type: Timing
Target: `sb_fe.c, sb_sw_lib.c`

Difficulty: High
Finding ID: TOB-SB-001

Description

In several places in the code, sensitive data is used in a conditional within an `SB_ASSERT`. Uses of `SB_ASSERT` are turned into C `assert` statements when `SB_DEBUG_ASSERTS` is enabled. Thus, a user risks revealing sensitive data via a timing side-channel attack if `SB_DEBUG_ASSERTS` is left enabled in production code.

An example appears in the function `sb_word_mask` in Figure 1.1. Short-circuiting will cause this function to execute fewer instructions and take less time when the argument a is 0.

```
// Returns an all-0 or all-1 word given a boolean flag 0 or 1 (respectively)
static inline sb_word_t sb_word_mask(const sb_word_t a)
{
   SB_ASSERT((a == 0 || a == 1), "word used for ctc must be 0 or 1");
   return (sb_word_t) -a;
}
```

Figure 1.1: `sb_fe.c#L88-L93.`

Exploit Scenario

Alice develops an embedded device and chooses Sweet B to provide its cryptographic capabilities. Alice thinks it is better to crash than to continue after an assertion violation, so she leaves `SB_DEBUG_ASSERTS` enabled in her production code. Eve discovers the mistake and steals Alice's clients' cryptographic material. Alice is forced to perform an expensive software update and/or product recall.

Recommendation

Short term, document the fact that `SB_DEBUG_ASSERTS` should not be enabled in production code.

Long term, produce a guide for developers on how to incorporate Sweet B into their projects. The guide should list which configurable defines are safe to enable in production code. The guide could also mention the types of errors that Sweet B checks for generally (e.g., whether a point is on a curve) and the types of errors that it ignores (e.g., whether a pointer is null). Such a guide would make it less likely for a developer to use Sweet B incorrectly.

4. HMAC_DRBG may lack backtracking resistance

Severity: Low
Difficulty: High
Type: Cryptography
Finding ID: TOB-SB-006
Target: `sb_hmac_drbg.h`, `sb_hmac_drbg.c`, `sb_sw_lib.h`, `sb_sw_lib.c`

Description
`HMAC_DRBG` is an implementation of a deterministic random bit generator (DRBG) specified in NIST SP 800-90. These random bit generators have a notion of security referred to as backtracking resistance (sometimes called forward secrecy), which guarantees that all outputs obtained prior to compromise of the DRBG state remain secure.

NIST SP 800-90 calls for the `HMAC_DRBG` implementation interface to include a `generate` function that generates pseudo-random bits. They further specify that this function should take as input, among other things, an optional "additional input." Sweet B complies with this standard and their `HMAC_DRBG` implementation allows this optional additional input.

Analysis by Woodage and Shumow identified that additional input is required for backtracking resistance in `HMAC_DRBG`, and it cannot be backtracking resistant without it.

If additional input is never supplied to the `generate` function, then `HMAC_DRBG` is not backtracking-resistant. If the DRBG state is compromised, then an attacker may possibly recover unseen outputs produced prior to the compromised state.

Although the authors of the paper believe this attack is infeasible, their claim is unproven and their justification is only sound when the HMAC is modeled as a random oracle. In short, they believe a successful attack is unlikely but they cannot formally prove this claim.

If additional input is *always* supplied to this function, then the `HMAC_DRBG` is backtracking-resistant. This analysis is performed in a conservative security model, so this positive result does not require the "additional input" to be pseudo-random.

Exploit Scenario
Alice develops an embedded device based on Sweet B. A memory unsafety vulnerability in the device allows leakage of `HMAC_DRBG` state. The attacker uses this issue to recover prior input and break perfect forward secrecy, thus recovering plaintext for intercepted encrypted traffic. Alternatively, the attacker recovers an ECDSA per-message secret and derives a long-term private key.

Recommendation

In the short term, document this limitation of `HMAC_DRBG` for users of the Sweet B library. Encourage users who want to formally guarantee backtracking resistance to always supply "additional input" to `generate` and consistently reseed their DRBG.

In the long term, consider strategies that:

- Expect the use of additional input and throw warnings or errors without it
- Abstract the use of additional input and seamlessly provide it for users
- Require users to explicitly opt-in to using `HMAC_DRBG` without additional input

Consider deprecating `HMAC_DRBG` and, instead, generating random data with `Hash_DRBG`. Woodage and Shumow's analysis proves that `Hash_DRBG` is a robust DRBG, and this robustness is not conditional on additional input.

References

- An Analysis of the NIST SP 800-90A Standard

5. Use of functions on the SDL List of Banned Functions

Severity: Informational
Type: Data Validation
Target: Various

Difficulty: Low
Finding ID: TOB-SB-002

Description
Sweet B makes use of functions that are on the Software Development Lifecycle (SDL) List of Banned Functions. Such functions are considered error-prone. Therefore, an alternative is recommended for each (see Table 2.1).

Banned function	**Recommended alternative**	**Brief summary of differences**
`memcmp()*`	`memcmp_s()`	For `memcmp_s`, a size argument accompanies both of its pointer arguments.
`memcpy()`	`memcpy_s()`	For `memcpy_s`, a size argument accompanies both its destination and source arguments.
`memmove()*`	`memmove_s()`	Like `memcpy`/`memcpy_s`.
`memset()`	`memset_s()`	For `memset_s`, a character count argument must be no more than the size of the destination buffer.
`strlen()*`	`strnlen_s()`	For `strlen_s`, an additional argument provides the maximum allowable length of the string.

Table 2.1: Functions on the SDL List of Banned Functions used by Sweet B.
An asterisk (*) indicates that the function is used only in testing code.

Recommendation
In place of each banned function in Table 2.1, consider using its recommended alternative. Doing so will increase confidence in the safety of Sweet B.

Western Digital noted that the recommended alternative functions in Table 2.1 are not available in the version of `libc` that they link against. Compiling those functions individually should be considered (see also TOB-SB-003). If this is not an option, we recommend switching to those functions if ever they become available.

References
- Intel SDL List of Banned Functions
- Microsoft adds memcpy to the SDL C/C++ banned API list

6. API for ECDSA signatures does not enforce secure message digests

Severity: Informational
Type: Cryptography
Target: `sb_sw_lib.h, sb_sw_lib.c`

Difficulty: N/A
Finding ID: TOB-SB-005

Description
The ECDSA signature algorithm takes as input a message and private key (as well as other information, such as curve parameters), and produces a signature. The algorithm calls for the message to first be digested by using a secure hash function.

The Sweet B API for signing messages using ECDSA (see Figure 5.1) takes as input the message digest. Therefore, the caller of this function is responsible for digesting the message to be signed with a secure hash function.

```
/** Signs the 32-byte message digest using the provided private key. If a \p
*  drbg is supplied, it will be used for the per-message secret generation
*  as per FIPS 186-4. The private key and message are used as additional
*  input to the \p drbg to ensure that the per-message secret is always
*  unique per (private key, message) combination. If no \p drbg is
*  supplied, RFC6979 deterministic secret generation is used instead.
...
*/
extern sb_error_t sb_sw_sign_message_digest(sb_sw_context_t context[static 1],
                                            sb_sw_signature_t signature[static 1],
                                            const sb_sw_private_t private[static 1],
                                            const sb_sw_message_digest_t
                                            message[static 1],
                                            sb_hmac_drbg_state_t* drbg,
                                            sb_sw_curve_id_t curve,
                                            sb_data_endian_t e);
```

Figure 5.1: `sb_sw_lib.h#L529-L557.`

This was discussed with Western Digital. Adjustments to the API were proposed; however, they did not entirely comply with the intended applications of this codebase. Ultimately, it was decided that it is the responsibility of the user to digest the messages securely.

Recommendation
Short term, update the comments surrounding `sb_sw_sign_message_digest` and similar functions to better inform users about securely digesting messages. Specifically, alert users that it is imperative that `sb_sha256` is used for digesting messages.

In addition to adding comments, add a function `sb_sha256` that combines `sb_sha256_init`, `sb_sha256_update`, and `sb_sha256_finish`, similar to how `sb_sw_sign_message_digest` combines `sb_sw_sign_message_digest_start`, `sb_sw_sign_message_digest_continue`, and `sb_sw_sign_message_digest_finish`. Also, add a function `sb_sw_sign_message` that would take as input a raw message, securely digest it using `sb_sha256`, and then pass it as input into the `sb_sw_sign_message_digest` function.

Long term, as additional digest functions are added to Sweet B (e.g., SHA-3, SHAKE256), add unit tests to verify their compatibility with `sb_sw_sign_message_digest`.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
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 protecting the privacy or integrity of 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
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Non-Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- Within sb_fe_armv7.s, the prototype for the function sb_fe_equal is inconsistently tabbed relative to other prototypes in that file.
- In some parts of the code, magic numbers are used. For example, several of the tests within sb_sw_lib_test.c.h have a loop of the following form:

```
do {
      ...
} while (i < 128);
```

  In at least one instance, a define is given, but does not appear to be used (SB_TEST_ITER_DEFAULT).
- A build should fail gracefully if __int128_t is not available. Checking whether __SIZEOF_INT128__ is defined may provide a means for doing so.
- There is a typo in an sb_fe.h comment: “beneit” should be “benefit.”
- The comments surrounding the function sb_hmac_drbg_generate_additional_vec in sb_hmac_drbg.h state that the error SB_ERROR_INPUT_TOO_LARGE is returned if the sum of the additional input lengths is greater than or equal to SB_HMAC_DRBG_MAX_ADDITIONAL_INPUT_LENGTH. This is inaccurate; this error is returned only if the sum of the additional input lengths is greater than SB_HMAC_DRBG_MAX_ADDITIONAL_INPUT_LENGTH (but not equal).

C. Instruction Trace Analysis

To help verify the constant time behavior of certain functions within Sweet B, Trail of Bits used a modified version of QEMU to obtain instruction traces for those functions. We verified that the instruction traces did not vary with the input to the function. We also checked those instruction traces for certain problematic instructions. This appendix details our methodology, discusses its limitations, and presents our results.

Methodology

We used a modified version of QEMU to obtain instruction traces for runs of an ARM executable. We wrote a tool to extract from those traces the portions specific to the functions of interest to us. We verified that the set of traces specific to any one function was a singleton by hashing each resulting trace and comparing the hashes. We further checked those traces for conditional loads and stores, and instructions with variable cycle counts other than branches.

We made two types of modifications to QEMU:

- Stock QEMU provides the option to dump assembly code for a translation block (TB) the first time it is seen.[1] We disabled one conditional within QEMU so it would dump assembly code for a TB every time it is seen.
- We eliminated some of QEMU's log output to make it more concise and easier to parse.

With our modifications in place, we could obtain a complete instruction trace for an ARM executable with a command like the following:

`qemu-arm -d in_asm,nochain` *executable arguments*

Sample output appears in Figure C.1. The example corresponds to the first TB of `sb_sw_point_multiply` in our build of `sb_test`.

---

[1] In QEMU, translation blocks (TBs) are constructed as follows. Suppose that PC is the current program counter and that I_{PC} is the instruction at PC. If execution of I_{PC} would always be followed by execution of the next instruction in memory, then I_{PC} is included in the current TB and the process continues with PC equal to the address of the next instruction in memory. If, on the other hand, execution of I_{PC} could be followed by execution of some instruction other than the next instruction in memory, then the current TB ends with I_{PC}, and, for each instruction that could be executed following I_{PC}, a new TB is constructed with PC equal to the address of that instruction. A branch would be an example of the latter kind of instruction. To our knowledge, each QEMU translation block is a "basic block" in the standard compiler sense.

```
0x00017848:  e92d 43f0  push.w   {r4, r5, r6, r7, r8, sb, lr}
0x0001784c:  4688       mov      r8, r1
0x0001784e:  4611       mov      r1, r2
0x00017850:  461a       mov      r2, r3
0x00017852:  4b1a       ldr      r3, [pc, #0x68]
0x00017854:  4c1a       ldr      r4, [pc, #0x68]
0x00017856:  447b       add      r3, pc
0x00017858:  b085       sub      sp, #0x14
0x0001785a:  591e       ldr      r6, [r3, r4]
0x0001785c:  9f0e       ldr      r7, [sp, #0x38]
0x0001785e:  6833       ldr      r3, [r6]
0x00017860:  9701       str      r7, [sp, #4]
0x00017862:  9303       str      r3, [sp, #0xc]
0x00017864:  9b0d       ldr      r3, [sp, #0x34]
0x00017866:  4605       mov      r5, r0
0x00017868:  9300       str      r3, [sp]
0x0001786a:  9b0c       ldr      r3, [sp, #0x30]
0x0001786c:  f7ff ff66  bl       #0x1773c
```

Figure C.1: The first translation block (TB) belonging to `sb_sw_point_multiply` *in our build of* `sb_test`*.*

Our tool to extract function-specific traces takes as input a list of pairs of the form (*address*, *path-prefix*):

- Each *address* is the start of a function of interest.
- Each *path-prefix* determines where traces for the associated function should be written.

The tool monitors an incoming stream of assembly instructions. When an instruction matches an *address* of interest, the following occurs. First, the address of the previous instruction is recorded; call this value PREV-PC. Second, an associated counter is incremented, and a file is opened at:

path-prefix '-' *counter*

Instructions are streamed out to that file until an instruction with address PREV-PC + 4 is seen. That's because when targeting the ARM, function calls are typically compiled into `bl` ("branch long") instructions, which take up four bytes (in both ARM and Thumb mode). Thus, if the instruction at PREV-PC was a `bl` instruction corresponding to a function call, the function will return to PREV-PC + 4. So once an instruction with address PREV-PC + 4 is observed, we can assume the function has returned.[2]

[2] One could contrive an executable for which this assumption does not hold, e.g., using recursion or tail calls. But the executables we are testing do not involve such trickery.

While instructions are being streamed out to a file, monitoring for *address*es of interest is disabled.

If *n* is the number of times that the function associated with some *path-prefix* was called, then when the tool finishes, one will have a set of files named:

path-prefix`-0`
path-prefix`-1`
path-prefix`-2`
...
path-prefix-n

One can verify that the files are all the same by verifying that they all have the same hash.

We also checked the instruction traces for conditional loads and stores. Although we found some, they all appeared within calls to `memcpy`. This, in turn, led to TOB-SB-003.

Finally, we checked the instruction traces for instructions with variable cycle counts other than branches—specifically, the instructions in Table C.1. Note that a branch is a variable cycle count instruction because it could require the instruction pipeline to be refilled. However, when this occurs, it will be reflected in the instruction trace. Thus, if two instruction traces match, the pipeline was refilled in both traces or in neither trace for every branch.

Instruction	Description
`SDIV`	Signed divide
`UDIV`	Unsigned divide
`CPSID`	Disable interrupts
`CPSIE`	Enable interrupts
`MRS`	Read special register
`MSR`	Write special register
`WFE`	Wait for event
`WFI`	Wait for interrupt
`ISB`	Instruction synchronization barrier
`DMB`	Data memory barrier
`DSB`	Data synchronization barrier

Table C.1: Instructions with variable cycle counts.

Limitations

While checking for conditional loads and stores, we noticed other conditional instructions besides branches. For example, the following instruction appeared in calls to multiple functions:

```
movhs    r3, r2
```

Our method does not capture whether such an instruction's condition held, i.e., whether the operation was performed. Put another way, we cannot rule out the possibility that such an operation was performed in one trace, but not performed in another (identical) trace.

Results

The short-Weierstrass operations in Table C.2 were verified to have identical instruction traces with no problematic instructions (as described above) for 1,197 randomly generated inputs. Similarly, the prime-field element operations in Table C.3 were verified to have identical instruction traces with no problematic instructions for 32,699 randomly generated inputs.[3] The tables also give the number of instructions for each trace in our build of our test program.

Function	**Number of instructions**
sb_sw_compress_public_key	4461
sb_sw_compute_public_key	2900665
sb_sw_decompress_public_key	142325
sb_sw_generate_private_key	284774
sb_sw_hkdf_expand_private_key	113959
sb_sw_invert_private_key	497459
sb_sw_point_multiply	2905571
sb_sw_shared_secret	2905261
sb_sw_sign_message_digest	3376735
sb_sw_valid_private_key	1131
sb_sw_valid_public_key	4310
sb_sw_verify_signature	3444946

Table C.2: Short-Weierstrass operations required to have constant time behavior.

[3] If these numbers seem unusual, it is because we ran the tests for as long as possible, as opposed to running them some predetermined number of times.

Function	Number of instructions
sb_fe_add	28
sb_fe_cond_add_p_1	49
sb_fe_cond_sub_p	36
sb_fe_ctswap	44
sb_fe_equal	29
sb_fe_lt	20
sb_fe_mod_add	198
sb_fe_mod_double	201
sb_fe_mod_inv_r	189616
sb_fe_mod_negate	185
sb_fe_mod_reduce	394
sb_fe_mod_sqrt	138624
sb_fe_mod_sub	181
sb_fe_mont_convert	404
sb_fe_mont_mult	401
sb_fe_mont_reduce	406
sb_fe_mont_square	404
sb_fe_sub	31
sb_fe_sub_borrow	29
sb_fe_test_bit	59

Table C.3: Prime-field element operations required to have constant time behavior.

D. Unit Test Coverage Analysis

Trail of Bits reviewed the Sweet B unit tests to identify possible gaps in coverage. Following our analysis, we recommend adding unit tests to exercise the following code:

- the loop within the body of `sb_fe_rshift`
- the error-handling code in `sb_hmac_drbg.c`
- two currently untested edge cases in `sb_sw_sign_continue` and `sb_sw_invert_private_key` within the file `sb_sw_lib.c`
- the prime-field operations listed in Table D.2 below

File	Lines	Lines not covered	% not covered
sb_fe.c	630	1	0.16
sb_fe_tests.c.h	442	6	1.36
sb_hkdf.c	259	0	0
sb_hmac_drbg.c	314	13	4.14
sb_hmac_sha256.c	313	0	0
sb_sha256.c	448	0	0
sb_sw_lib.c	2329	8	0.34
sb_sw_lib_tests.c.h	1696	0	0
sb_test.c	157	36	22.93
sb_test_cavp.c	602	8	1.33
sb_test_list.h	116	0	0

Table D.1: Raw unit test coverage results.

As seen in Table D.1, about half of the files (5 of 11) are covered completely. We address the remaining files individually.

- `sb_fe.c`: The one unexecuted line is part of an `SB_ASSERT` statement. The fact that this line is unexecuted is a demonstration of correct behavior.
- `sb_fe_tests.c.h`: The six unexecuted lines are in the body of a loop in `sb_fe_rshift`. We recommend adding one or more unit tests to exercise that loop.
- `sb_hmac_drbg.c`: The thirteen unexecuted lines are related to error-handling. It is considered good practice to test outside the "happy path," so we recommend adding one or more unit tests to exercise that error-handling code.
- `sb_sw_lib.c`: The eight unexecuted lines concern edge cases in the functions `sb_sw_sign_continue` and `sb_sw_invert_private_key`. We recommend adding one or more unit tests to exercise those edge cases.
- `sb_test.c`: The 36 unexecuted lines concern failing test cases (of which there were none), handling of improper command line arguments, or alternative testing modes. Thus, these unexecuted lines are not relevant.
- `sb_test_cavp.c`: The eight unexecuted lines involve error-handling within the code that reads in the CAVP vectors. The files that contain those vectors are considered static, so these unexecuted lines are not relevant.

We noticed that several of the prime-field operations (i.e., the functions declared in `sb_fe.h`) are not tested directly within `sb_fe_tests.c.h`. These functions are tested indirectly, e.g., via the tests in `sb_sw_lib_tests.c.h`. However, in the interest of completeness, we recommend that at least one unit test be devoted to each of the prime-field operations not currently tested in `sb_fe_tests.c.h`.

Prime-field operation	**Tested in `sb_fe_tests.c.h`?**
`sb_fe_add`	Yes
`sb_fe_cond_add_p_1`	No
`sb_fe_cond_sub_p`	No
`sb_fe_ctswap`	No
`sb_fe_equal`	Yes
`sb_fe_lt`	No
`sb_fe_mod_add`	No
`sb_fe_mod_double`	No
`sb_fe_mod_inv_r`	No
`sb_fe_mod_negate`	No
`sb_fe_mod_reduce`	No
`sb_fe_mod_sqrt`	Yes
`sb_fe_mod_sub`	No
`sb_fe_mont_convert`	No
`sb_fe_mont_mult`	Yes
`sb_fe_mont_reduce`	Yes
`sb_fe_mont_square`	Yes
`sb_fe_sub`	Yes
`sb_fe_sub_borrow`	No
`sb_fe_test_bit`	No

Table D.2: Prime-field operations that are tested directly.

E. Build Configuration Analysis

To identify ways in which Sweet B could be misconfigured, Trail of Bits enumerated multiple possible build configurations and ran the unit tests under each. This analysis led to TOB-SB-004.

The exact build parameters tested, and the values used for each, appear in Table E.1. In total, 1,152 build configurations were tested. Aside from the issues discussed in TOB-SB-004, the unit tests passed in each configuration.

Parameter	**Values tested**
`SB_WORD_SIZE`	1, 2, 4
`SB_FE_VERIFY_QR`	Undefined, 1
`SB_FE_ASM`	Undefined, 1
`SB_DEBUG_ASSERTS`	Undefined, 1
`SB_UNROLL`	0, 1, 2, 3
`SB_HMAC_DRBG_RESEED_INTERVAL`	14 (min), 1024 (default)
`SB_HMAC_DRBG_MAX_BYTES_PER_REQUEST`	128 (min), 1024 (default), 65536 (max)
`SB_HMAC_DRBG_MAX_ENTROPY_INPUT_LENGTH`	256 (min), 1024 (default)

Table E.1: Build parameters and the values tested.

Two points are worth mentioning:

- We did not test build configurations with `SB_WORD_SIZE` equal to 8. To build with such a configuration, the type `__uint128_t` must be available. However, to our knowledge, there is no widely available compiler for the ARM Cortex-M4 that provides this type.
- To work around TOB-SB-004, we disabled the assembly version of `sb_fe_mont_mult` for this analysis. Thus, even when `SB_FE_ASM` was enabled, the C version of `sb_fe_mont_mult` was used. Use of the C version allowed the unit tests to pass.

Note that once proper remediations for TOB-SB-004 have been implemented, many of the build configurations described above will become invalid. The following are two examples:

- `SB_FE_ASM` = 1 and `SB_WORD_SIZE` = 2
- `SB_FE_ASM` = 1 and `SB_FE_VERIFY_QR` = 1