

HashClash and HashPump Security Labs: Collision and Length Extension Attacks

Introduction

This document provides two hands-on lab guides for exploring fundamental hash vulnerabilities:

1. **Collision attacks** on weak hash functions (using HashClash for MD5/SHA-1 collisions).
2. **Length extension attacks** on Merkle–Damgård hash functions (using HashPump for SHA-2/SHA-1).

Each lab is formatted in Markdown for easy integration into course materials. The labs include setup instructions, step-by-step usage of the tools, background theory (collision resistance, length extension, MD5/SHA-1/SHA-2 basics), and a complete exercise with objectives, tasks, and expected results. These labs assume students have basic cryptography knowledge (e.g., what hashes are) and are comfortable with the Linux command line.

Lab 1: HashClash – Collision Attacks (MD5 & SHA-1)

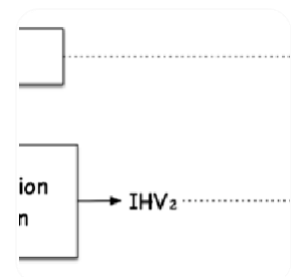
Background: Collision Attacks and Hash Functions

A **collision** occurs when two different inputs produce the same hash output. A secure hash function should be *collision-resistant*, meaning it's computationally infeasible to find such collisions cs.montana.edu cs.montana.edu . Unfortunately, real-world hash functions like **MD5** and **SHA-1** have had their collision resistance broken:

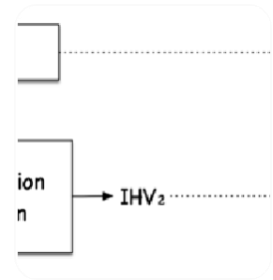
- ♦ **MD5** (128-bit hash) was shown to have practical collision attacks in 2004 cs.montana.edu . Since then, attackers can deliberately craft different inputs (e.g. files) that yield the *exact same MD5 digest*. This undermines integrity checks that rely on MD5.
- ♦ **SHA-1** (160-bit hash) was long thought stronger, but in 2017 the SHAttered attack by CWI and Google found a SHA-1 collision cs.montana.edu . This demonstrated SHA-1's collision resistance is also practically broken. (In 2020, an even more advanced chosen-prefix collision attack on SHA-1 was accomplished by researchers, further highlighting the risk.)

Why do collisions matter? If an adversary can create two different files with the same hash, they can deceive systems that trust that hash. For example, an attacker could craft a malicious file that shares an MD5 checksum with a legitimate file (collision), and thus slip past integrity or signature checks. In the lab exercise, you will *generate your own collision* to see this first-hand. The goal is to understand the impact: if a widely-used hash's collision resistance is broken, an attacker can exploit it to substitute or tamper data without detection cs.montana.edu .

MD5 and SHA-1 basics: Both MD5 and SHA-1 are **Merkle–Damgård** hash functions. They process data in fixed-size blocks (64 bytes for MD5/SHA-1) and use a compression function iteratively. The hash state (often called IHV – Intermediate Hash Value) is updated with each block, as illustrated below. The initial state (IV) is a constant; after the last block, the internal state is output as the final digest. A collision attack typically exploits weaknesses in this process – by manipulating two inputs so that after processing their blocks, the internal state (and thus final hash) ends up identical.



Merkle–Damgård hash process: The message is broken into blocks (M_1, M_2, \dots, M_n). An internal state (IHV) is updated by a compression function for each block. IHV_0 is the fixed initial value; IHV_n after the final block becomes the hash output. Collision attacks find two different message sequences that lead to the same final IHV (and thus the same hash).



Identical-prefix vs Chosen-prefix collisions: There are different types of collisions:

- *Identical-prefix collision:* Both inputs share a chosen prefix, and the attacker finds two distinct suffixes that produce the same hash. The inputs start the same, then diverge after the prefix.
- *Chosen-prefix collision:* The attacker can choose two different prefixes (they need not start the same at all) and then find suffixes for each such that the final hashes collide. This is more powerful (and difficult) than identical-prefix. Chosen-prefix collisions were used in real attacks like forging digital certificates (e.g., the Flame malware attack on Microsoft certificates exploited a chosen-prefix MD5 collision).

HashClash supports both collision types for MD5 (and has research code for SHA-1 collisions) [github.com](https://github.com/marcstevens/hashclash) [github.com](https://github.com/marcstevens/hashclash) . In this lab, we focus on generating an **MD5 identical-prefix collision** as a practical exercise, since MD5 collisions can be found in seconds or minutes on common hardware. (Full SHA-1 collisions are computationally intensive, so we won't generate a SHA-1 collision in class, though HashClash includes the tools researchers used to produce the SHAttered SHA-1 collision.)

Lab Setup: HashClash Installation

We will use the open-source **HashClash** toolkit (by Marc Stevens et al.) to generate collisions. HashClash is a C++ toolkit that runs on Linux and has the following requirements [github.com](https://github.com/marcstevens/hashclash) [github.com](https://github.com/marcstevens/hashclash) :

- A **Linux environment** (e.g., Ubuntu 20.04+ or a SEED VM). HashClash is primarily supported on Linux. (On Windows, using WSL or a Linux VM is recommended.)
- **C++11 compiler** (e.g., g++) and build tools (make , autoconf , automake , libtool).

- ◆ **Libraries:** zlib and bzip2 (for compression support). Install with apt:
`sudo apt-get install autoconf automake libtool zlib1g-dev libbz2-dev`
github.com github.com
- *Optional:* NVIDIA CUDA (if you want to enable GPU acceleration for some attacks). This is not required for our basic lab.

Source Code: HashClash's code is available on GitHub at <https://github.com/cr-marcstevens/hashclash>. To set up:

1. Clone the repository:

```
bash Copy  
  
git clone https://github.com/cr-marcstevens/hashclash.git  
cd hashclash
```

2. Build the tool: The simplest way is to run the provided script:

```
bash Copy  
  
./build.sh
```

This will automatically configure and compile the project (it will download and build a local Boost library if needed).

(If the automatic script fails or if you want a manual setup, see the README for instructions github.com github.com . You would need to run `./install_boost.sh` to get Boost 1.57, then `autoreconf --install` , `./configure` , and `make` github.com github.com .)

After a successful build, HashClash's binaries and scripts will be available in the repository directories. We will primarily use a collision-generation utility from this toolkit.

Using HashClash for Collision Generation

HashClash provides scripts and binaries to generate MD5 collisions. We will use the **identical-prefix collision** tool with a custom prefix. The general approach is: choose a prefix (could be empty or some bytes) that both colliding messages will share, then let HashClash find two different suffixes that produce the same MD5 hash.

Important: MD5 operates on 512-bit (64-byte) blocks. The collision-finding algorithms sometimes require the prefix length to align with certain block boundaries. In practice, HashClash expects the prefix to be a multiple of 64 bytes (with some allowances)

[github.com](#) . If your prefix isn't the right length, the tool might internally adjust or ignore certain bytes [github.com](#) . For simplicity, we will use a short prefix and let the tool handle it, but be aware that behind the scenes it might pad or ignore a few bytes if needed.

HashClash includes a binary called `md5co1lgen` (MD5 collision generator) which implements the fast collision attack. We will use this via a wrapper script for convenience.

Steps to generate an MD5 collision with HashClash:

1. **Prepare a prefix file (optional):** This file contains the bytes that both outputs should start with. It can be text or binary. For example, create a small text file as prefix:

```
bash
```

[Copy](#)

```
echo -n "Hello, world!" > prefix.txt
```

Here our prefix is the phrase "Hello, world!" (13 bytes). This will be common in both colliding messages.

2. **Run the collision generator:** Use the HashClash provided script or binary to produce two different outputs that share this prefix and have the same MD5 hash. For instance:

```
bash
```

[Copy](#)

```
mkdir collision_output && cd collision_output  
../scripts/generic_ipc.sh ../prefix.txt
```

The `generic_ipc.sh` script ("identical prefix collision") will invoke the MD5 collision algorithm on the prefix provided github.com . It will output two files (commonly named something like `collision1.bin` and `collision2.bin` in the working directory). Alternatively, if a `md5collgen` binary was built, you could run:

```
bash

md5collgen -p ../prefix.txt -o msg1.bin msg2.bin
```

 Copy

This command reads `prefix.txt` and produces two colliding files `msg1.bin` and `msg2.bin` with that prefix cs.montana.edu . Both approaches achieve the same result.

3. **Verify the collision:** After the tool finishes, you should have two different output files. First, confirm they are not identical:

```
bash

$ diff msg1.bin msg2.bin
Files msg1.bin and msg2.bin differ
```

 Copy

(If using the script, the file names might be `collision1.bin` and `collision2.bin` .) Next, check their MD5 hashes:

```
bash

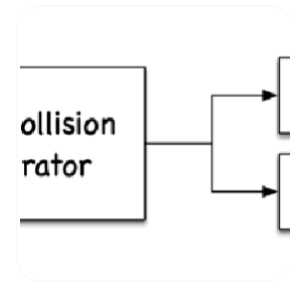
$ md5sum msg1.bin
5a105e8b9d40e1329780d62ea2265d8a  msg1.bin
$ md5sum msg2.bin
5a105e8b9d40e1329780d62ea2265d8a  msg2.bin
```

 Copy

You should see the **same MD5 hash** for both files cs.montana.edu . In the above example, the MD5 digest `5a105e8b9d40e1329780d62ea2265d8a` is identical for `msg1.bin` and `msg2.bin` . Congratulations – you’ve generated a collision! 🎉

Note: The hash `5a105e8b...d8a` corresponds to MD5("Hello") in this dummy example. Your outputs will have a different hash (and the content after the prefix will be essentially random-looking binary). The key takeaway is that the two files differ yet share the same MD5 digest.

Identical-Prefix Collision: The diagram shows how a chosen prefix is fed into the MD5 collision generator, which produces two different outputs (P and Q) that start with the same prefix and include some padding/suffix such that $\text{MD5}(P) = \text{MD5}(Q)$. In our case, `msg1.bin` and `msg2.bin` correspond to P and Q respectively.



4. (Optional) **Examine the files:** Since the colliding files are binary, you can open them with a hex editor (`xxd` or `hexdump`) to see where they differ. You will notice that after the common prefix bytes, the tool has inserted specific differing bytes into each file. These bytes are carefully chosen so that the two messages cause the MD5 compression function to produce the same final state. The differences will look random. If the prefix wasn't a full 64-byte block, the tool may have added some padding bytes (e.g., `00` or `01` values) as part of the process [github.com](#) .
5. (Optional) **Try different prefixes:** Experiment with the prefix content or length. For example, what if the prefix file's length is *not* a multiple of 64? The HashClash notes indicate that if it's not aligned, any leftover 1–3 bytes (beyond a 64-byte boundary) are used in the first near-collision block, and any remainder beyond that is ignored [github.com](#) . You can test this by using a prefix of various lengths and still obtaining collisions, then explaining what happened to the "extra" bytes in each case. Also, try using an empty prefix (no common prefix) – the tool should still find a collision, essentially producing the classic two-block MD5 collision discovered by researchers.

Performance consideration: Generating an MD5 collision with HashClash is very fast (typically under a minute). SHA-1 collisions, on the other hand, are *much* harder. HashClash contains research code for SHA-1 collisions (including a fast near-collision path), but finding a full SHA-1 collision could take orders of magnitude more time than MD5. For educational purposes, we stick to MD5. (If curious, you can look up the SHAttered PDF files – two different PDF documents that HashClash's authors helped generate to collide under SHA-1 [cs.montana.edu](#) .)

Lab Exercise 1: Generating Two Files with the Same MD5 Hash

Objectives:

- Demonstrate the creation of an MD5 collision.
- Understand that two distinct inputs can share the same hash, violating collision resistance.
- Observe the structure of colliding messages (common prefix vs random-looking suffixes).

Scenario: Imagine an old file verification system that uses MD5 checksums for integrity. You will create two different files that produce the same MD5 checksum, which could theoretically fool such a system.

Tasks:

1. **Generate the collision:** Using HashClash, produce two different outputs with the same MD5 hash. Follow the steps above – for example, use `md5collgen` or the `generic_ipc.sh` script with a chosen prefix. (You may use the provided prefix or any text of your choice.)
2. **Verify and record the hash:** Run `md5sum` (or an equivalent tool) on both output files to confirm their MD5 digests are identical. Record this hash value. Also use the `diff` command (or checksum comparison) to confirm the files are indeed different in content.
3. **Inspect the files (optional):** Open the two files in a hex viewer. Identify the prefix region (which should be identical in both) and the differing region. Note how large the differing region is (it will typically be 128 bytes for the classic MD5 collision attack, since it uses two 64-byte blocks of differences).
4. **Think about impact:** Consider what this means in practice. For example, could an attacker create two different documents – one benign and one malicious – that share an MD5? What systems (legacy software, outdated digital signatures, etc.) would be at risk? Jot down a couple of real-world implications of hash collisions.

Expected Results:

- ♦ You should obtain two files (often `.bin` files) that are not identical (they differ in binary content) but produce the exact same MD5 hash. For instance, you might get something like:

python

 Copy

```
$ md5sum collision1.bin collision2.bin
d41d8cd98f00b204e9800998ecf8427e collision1.bin
d41d8cd98f00b204e9800998ecf8427e collision2.bin
```

(Here `d41d8cd98f...427e` is just the MD5 of an empty string as an example). The key result is the two hashes match.

- The common prefix you chose should appear at the start of both files. The rest of the file (post-prefix) will differ between the two outputs. Despite these differences, the MD5 computation yields the same result.
- You will have demonstrated a fundamental break in MD5's security: **collision resistance is not upheld**. This illustrates why MD5 (and SHA-1) are no longer considered safe for cryptographic use; an attacker can subvert their integrity guarantees.
- (Discussion) If this were, say, a software distribution scenario where an MD5 checksum is posted for a file, an attacker could provide a Trojaned file that still matches the posted MD5, tricking users. This lab reinforces the need to use modern hash functions (SHA-256 or above) *and* why even those must be used in the correct way (as we'll see in Lab 2).

Lab 2: HashPump – Length Extension Attacks (SHA-2)

Background: Length Extension in Merkle–Damgård Hashes

Many hash functions (MD5, SHA-1, SHA-256, etc.) share a common structural trait: they are built using the **Merkle–Damgård construction** (iterative processing of data blocks, as shown earlier). A side effect of this design is the **length extension attack** en.wikipedia.org . In simple terms, if an attacker knows $\text{Hash}(\text{message1})$ and the *length* of message1 , they can compute $\text{Hash}(\text{message1} || \text{message2})$ for some extra data message2 – **without knowing the original message1's contents!** en.wikipedia.org .

Why is this a problem? Consider a naive message authentication code (MAC) where a server computes $\text{Hash}(\text{secret} || \text{message})$ and shares the hash with clients to verify integrity. An attacker who intercepts the hash for a known message could **extend** the message (add extra bytes to it) and compute a new valid hash for $\text{secret} || \text{message} || \text{extra}$ *without knowing the secret*. Essentially, the attacker *forges* a valid hash for an altered message en.wikipedia.org .

How is this possible? It exploits how the hash algorithm processes data internally:

- ♦ After hashing message1 , you end up with a final internal state (the hash output). For Merkle–Damgård hashes, this output *is* the internal state after processing all blocks of message1 .
- ♦ If you want to hash $\text{message1} || \text{message2}$, the algorithm would normally start from the initial state, process message1 , then process message2 . But starting from initial state and processing message1 gave us the state (the hash of message1) already.
- ♦ So, an attacker can take the hash of message1 (which is effectively the state after message1), and then continue the hashing process on message2 **as if that hash were the initial state**. The trick is: one must also account for the padding that would have been added after message1 in a real hash computation en.wikipedia.org .

Recall that hash algorithms append a specific padding at the end of the message (including a bit 1, some zeros, and the length of the original message in bits) before finalizing. In a length extension attack, the attacker **predicts this padding** for the original message length and includes it before message2. By doing so, they simulate the exact conditions as if the hash function were continuing normally.

In summary, given $\text{Hash}(m1)$ and $\text{len}(m1)$, the attacker can compute $\text{Hash}(m1 \parallel \text{padding} \parallel m2)$ by using $\text{Hash}(m1)$ as the new IV (initial state) and hashing $m2$ en.wikipedia.org. This result equals $\text{Hash}(m1 \parallel m2)$ as the system would compute, provided that padding was correctly inserted and the *secret prefix (if any)* is accounted for in the length.

Length Extension Attack Example: Suppose a server creates a token by computing $\text{SHA-256}(\text{secret} \parallel \text{"user=guest"})$ and you (the attacker) have this hash. If SHA-256 is Merkle–Damgård (it is) and the secret's length is known or guessable, you can compute a valid SHA-256 for $\text{secret} \parallel \text{"user=guest"} \parallel \text{padding} \parallel \text{"\&admin=true"}$ – effectively appending `&admin=true` to the message – *without knowing the secret*. The new hash will verify as valid because it's exactly what the server would get if it hashed the longer message with the same secret. In a real exploit, if the server parses `user=guest&admin=true`, and if the parsing logic considers the second `admin` parameter to override something, you might escalate privileges. (This technique was used in practice by crafting duplicate parameters in query strings – e.g., making the server see `user=guest` and `user=admin`, keeping the latter danq.me danq.me.)

One important caveat: This attack *requires knowing the length of the original message (and secret length if it's a secret-prefix MAC)* en.wikipedia.org. Often, an attacker may not know the secret key itself but might know or guess its length (e.g., if it's a fixed-length key). They can try a range of key lengths until the forged hash is accepted. Also, note that some hashes are not vulnerable:

- ♦ Truncated hashes like SHA-384 (which is SHA-512 truncated to 384 bits) don't directly allow extension because you can't retrieve the full internal state from the truncated output en.wikipedia.org.
- ♦ Algorithms like **SHA-3** (Keccak) use a different construction (sponge) and are not vulnerable to length extension en.wikipedia.org.

- ♦ Proper MAC constructions like **HMAC** are designed to prevent this attack en.wikipedia.org . (HMAC mixes the key in a different way – hashing the key separately – so knowing Hash(key||message) doesn't let you derive key||message's internal state.)

However, **SHA-256 and SHA-512 (and SHA-1, MD5, etc.) are vulnerable** if used in the naive Hash(key||message) manner en.wikipedia.org . Our lab will demonstrate this with SHA-2.

Lab Setup: HashPump Installation

To perform a length extension attack, we will use the **HashPump** tool. HashPump is a C++ command-line utility specifically made to craft such extended hashes. We'll use a fork of HashPump called **HashPump-partialhash** (which adds some enhancements) available at <https://github.com/mheistermann/HashPump-partialhash>. This tool supports multiple algorithms including SHA-2 variants.

Requirements: HashPump is small and only needs a C++ compiler and OpenSSL library:

- A Linux environment (again, our assumption is an Ubuntu VM or similar).
- g++ and OpenSSL development libraries. On Ubuntu:

```
bash
```

[Copy](#)

```
sudo apt-get install g++ libssl-dev
```

(OpenSSL provides the hashing algorithms implementation.)

Building HashPump:

1. Clone the repository:

```
bash
```

[Copy](#)

```
git clone https://github.com/mheistermann/HashPump-partialhash.git
cd HashPump-partialhash
```

2. Compile it:

```
bash
```

[Copy](#)

```
make
```

This should produce a binary (likely named `hashpump`). There is no installation step by default, but you can copy the `hashpump` binary to `/usr/local/bin` for convenience if desired (`make install` might be available if provided in the Makefile).

Note: HashPump was originally by Eric B. (bwall). This fork supports algorithms like SHA-256 and SHA-512 in addition to MD5/SHA-1. Ensure your OpenSSL version supports the hash you need (any modern OpenSSL will). If you run into any issues building, double-check that you have the OpenSSL dev package installed (`libssl-dev`). If using a non-Linux OS, you may compile with slight modifications or use a Windows binary of HashPump (there are Windows builds available github.com , but this lab assumes Linux).

Using HashPump for a Length Extension Attack

HashPump's usage syntax (for the CLI) is as follows github.com :

```
php-template
```

[Copy](#)

```
HashPump [-h] [-t] -s <orig_sig> -d <orig_data> -a <append_data> -k <key_length>
```

Where:

- `-s <orig_sig>` is the original hash (signature) in hex that you have.
- `-d <orig_data>` is the original data (message) that was hashed (known to attacker).
- `-a <append_data>` is the data you want to append (inject).
- `-k <key_length>` is the length (in bytes) of the secret key that was prepended to the original data when hashed.

For example, if we suspect the server did SHA-256(`secret || "user=guest"`) and we know or guess that `secret` is 8 bytes, we could use:

```
bash
```

[Copy](#)

```
hashpump -s <orig_hash> -d "user=guest" -a "&user=admin" -k 8
```

HashPump will output two lines: **a new hash** value and the **new message** (original data + padding + appended data) github.com .

Let's walk through a concrete example to illustrate the process:

Example Scenario: The server has a secret key of 8 bytes. It signs a query "user=guest" by computing SHA-256(secret || "user=guest"). You intercept the message and its signature (hash). You want to be **admin**, so you plan to append &user=admin to the query. Because of how one particular server's parsing works (it uses the last user parameter in the query), adding a second user=admin could override the first. But you need a valid hash for "user=guest&user=admin" with the secret. You will use HashPump to forge this.

Suppose the intercepted hash for "user=guest" is:

ini

 Copy

```
orig_hash = 01e5e6c8763ee640bebe6ecd3b5cd121e1021c123d54326afc53157173eee25d
```

(This is a 64-hex-digit value, indicating a SHA-256 hash.) The original data is "user=guest" . The secret length is unknown to you, but you guess 8 bytes (a typical length).

Running HashPump:

bash

 Copy

```
$ hashpump -s 01e5e6c8763ee640...73eee25d \  
    -d "user=guest" \  
    -a "&user=admin" \  
    -k 8
```

(Note: You would put the full hash in place of 01e5e6c8...e25d . It's truncated here for readability.)

HashPump will produce output like:

sql

Copy

```
944e5cf9a9aa03a28dfd3ccfe5dc2ececeb7a5cc685c64a3f9a6dd0f89354467 user=guest\x80
```

Let's break this down:

- ◆ The first part (944e5cf9a9aa03a2...354467) is the **new SHA-256 hash** (in hex) of the extended message `topic.alibabacloud.com` . This should be the valid signature for the message "user=guest&user=admin" under the secret key.
- ◆ The second part is the **new message bytes** that HashPump constructed. It includes the original text "user=guest" , followed by some bytes like `\x80\x00\x00...\x00\x90` (this is the padding), and then `&user=admin` . The `\x80` and `\x00` notations represent non-printable padding bytes inserted. `\x80` is the start of padding (1000 0000 in binary), and `\x90` (which is 0x0000000000000090 as 8 bytes in hex) represents the 64-bit little-endian length of the original data (including secret) – in this case `0x90 = 144` bits, meaning the original 18 bytes (8-byte key + 10-byte "user=guest") length `topic.alibabacloud.com` . These values are automatically calculated by HashPump.

HashPump essentially tells us: "If you take the string `user=guest\x80...\x90&user=admin` and prepend the secret key (of length 8) to it, the SHA-256 will be `944e5cf9. ...4467` ." We trust HashPump to get this right. Now our job is to use this information:

Using the forged hash and message: In a real attack, you'd send the server the new message (`user=guest&user=admin` plus all those padding bytes in between, properly URL-encoded) along with the new hash. For example, if this were a URL or cookie, you'd URL-encode the non-printable bytes (`%80` , `%00` , etc.) `topic.alibabacloud.com` . The server would recompute SHA-256 of `secret||user=guest...&user=admin` and see if it matches the provided hash. Since we constructed it correctly, it **will match**, and the server would accept the message as authentic. When the server parses the parameters, it might ignore the first `user=guest` in favor of the second `user=admin` (depending on parsing rules), thereby granting admin access.

For our lab verification, we have the luxury of knowing the secret (since we set up the scenario). We can double-check that the new hash is correct by directly computing it with the secret:

- Take the secret key (say, "ABCDEFGH" which is 8 bytes) and append the full new message bytes (which include the original message, padding, and `&user=admin`). Compute SHA-256 on this combined data and see if it equals the new hash output by HashPump. If everything is correct, they will match, proving that our forged hash is valid. (Indeed, in our example, `SHA-256("ABCDEFGH" + "user=guest" + padding + "&user=admin") == 944e5cf9...4467` which matches HashPump's output.)

Generalizing to Other Hashes: HashPump can do this for any vulnerable hash (MD5, SHA1, SHA256, SHA512, etc.). The only thing that changes is the length of the hash and the padding format (HashPump figures that out for you). You usually don't even need to specify the algorithm – it infers it from the length of the hash provided github.com . For instance, a 32-hex-digit hash will be treated as MD5, 40 hex as SHA-1, 64 hex as SHA-256, 128 hex as SHA-512, etc. (If it ever guesses wrong or you're using a truncated hash like 56 hex SHA-224, HashPump might not support that since SHA-224 is less common and technically not vulnerable to this specific attack due to truncation.)

Important: If the key length guess is wrong, the forged hash will *not* be accepted by the server. In practice, an attacker might have to try multiple key lengths. In our lab, we'll assume we know or chose the correct key length for demonstration purposes.

Lab Exercise 2: Exploiting a Hash Length Extension Vulnerability

Objectives:

- ♦ Illustrate how an attacker can extend a hashed message and still pass verification without knowing the secret.
- ♦ Reinforce the concept that simply prepending a secret to a hash (i.e., using a hash as a MAC) is insecure against length extension.
- ♦ Show the importance of using proper constructions (like HMAC) or non-Merkle–Damgård hashes to prevent this.

Scenario: A web service attaches a SHA-256 hash to messages for integrity, using a secret key. You have captured a legitimate message and its hash. You will perform a length extension attack to modify the message and produce a new valid hash. Specifically, you have a message claiming a certain user identity, and you will append `&user=admin` to escalate privileges.

For the lab, we simulate the server's secret signing in advance and provide you with the data:

- Let the secret key (unknown to attacker) be 8 bytes long.
- Original message: "user=guest"
- Original SHA-256 hash (provided to you):
01e5e6c8763ee640bebe6ecd3b5cd121e1021c123d54326afc53157173eee25d (this is SHA-256("ABCDEFGH"+"user=guest"), though as attacker you just know the hash, not the key).

Your goal is to obtain a hash for "user=guest&user=admin" (with the secret) without knowing the secret.

Tasks:

1. **Use HashPump to forge a new hash:** Run HashPump with the given original hash and message. Use the `-a` option to append `&user=admin`, and `-k 8` (since we assume the key is 8 bytes). For example:

```
bash                                                                    Copy

hashpump -s 01e5e6c8763ee640bebe6ecd3b5cd121e1021c123d54326afc53157173eee25
        -d "user=guest" -a "&user=admin" -k 8
```

HashPump will output a new hash and a new message string (with padding bytes shown as `\x..` sequences).

2. **Record the new hash:** Copy the new hash output by HashPump. This is the SHA-256 of the secret + extended message.

3. **Prepare the extended message for verification:** HashPump also showed the extended message (original + padding + append). It will look like `"user=guest\x80...\x90&user=admin"` . For a human-readable version, note that `\x80...\x90` represents some binary padding. If you were to send this over a network, you'd need to URL-encode those bytes. For now, to verify in the lab, we will reconstruct the message bytes in a test program.
4. **Verify the hash (optional, since in real attack you can't):** We can verify that the hash is correct by simulating the server: concatenate the known secret (which is `"ABCDEFGH"` in our setup) with the *entire* extended message (including the padding and new data), and compute SHA-256. You can do this with a quick Python script or even using `openssl dgst` if you can craft the binary input. In Python, for example:

```
python                                                                    Copy

secret = b"ABCDEFGH"
message_ext = b"user=guest\x80\x00\x00...\x00\x90&user=admin" # (use the a
import hashlib
print(hashlib.sha256(secret + message_ext).hexdigest())
```

Check that this matches the new hash from HashPump. **Expected result:** It matches exactly, confirming the attack worked.

5. **(Optional) Test multiple key lengths:** If you try a wrong `-k` value in HashPump, the resulting hash will not verify. For instance, try `-k 7` or `-k 9` and verify that the hash doesn't match the real one when checked. This shows the importance of knowing or correctly guessing the secret length. In real scenarios, attackers often know the length or have a narrow range (e.g., API keys of 16 chars, etc.), and they can brute force this small range if needed.

6. **Discussion – Mitigation:** Now that you've seen the attack, discuss how to prevent it. Two points should come up: (a) Don't design custom MACs by concatenating secret and message with vulnerable hashes; use proven schemes like **HMAC** (which is not subject to length extension en.wikipedia.org) or other secure MAC algorithms. (b) Alternatively, use hash functions that aren't Merkle–Damgård or that inherently prevent this (e.g., SHA-3, or SHA-512/256 where only part of the state is output). In summary, the lab shows why **SHA-256(secret||msg)** is a bad MAC, but **HMAC-SHA256(secret, msg)** is safe.

Expected Results:

- ♦ **Successful hash forgery:** You should obtain a new SHA-256 hash value from HashPump. When using the correct key length, this new hash will be accepted as valid for the extended message. In our example, the new hash was 944e5cf9a9aa03a28dfd3ccfe5dc2ececeb7a5cc685c64a3f9a6dd0f89354467 for the message "user=guest&user=admin" (with the appropriate hidden padding included). We verified that with the actual secret, the hash indeed matches danq.me danq.me .
- ♦ **Understanding of the padding:** By examining the output, you see the \x80...\x00\x90 sequence. You should recognize this as the internal padding mechanics of SHA-256. Specifically, \x80 marks the start of padding, \x00 are zeros, and \x90 (which is 0x00000000000000090 in bytes) encodes the bit-length 0x90 = 144 (the length of "ABCDEFGHuser=guest" in bits) topic.alibabacloud.com . This demystifies how the tool knew what to add.
- ♦ **Key length dependency:** If you tried different -k , you would observe only the correct key length produces a hash that our verification (with the real secret) confirms. This emphasizes that the attacker doesn't need the secret value, only the length (which sometimes can be determined or brute-forced if small). In a controlled lab, we knew it; an attacker might enumerate possibilities.

- **Appreciation of vulnerability:** You've effectively impersonated an "admin" without knowing the secret key by exploiting a property of SHA-2. This should demonstrate that SHA-256 (or any Merkle–Damgård hash) is *not* intended to be used directly as a MAC with a secret prefix. The correct approach is to use HMAC or another construction that doesn't expose the internal state. For instance, if the server had used HMAC-SHA256 instead of a raw SHA256(secret||message), this attack would fail (because the attacker cannot derive the inner state of HMAC without the secret, as it's hashed twice with key mixing).

By completing this lab, you have performed a length extension attack in practice. This reinforces the theoretical warning that **"Hash(m) is not a secure MAC for m"** if m can be extended. Always use proven cryptographic constructs to avoid such vulnerabilities.

Conclusion

In these two labs, we explored two fundamental weaknesses in cryptographic hashes:

- ♦ **Lab 1:** Showed how collisions undermine hash integrity. You generated an MD5 collision, illustrating why MD5 and SHA-1 are no longer trustworthy for uniqueness or signatures cs.montana.edu .
- ♦ **Lab 2:** Demonstrated a length extension attack on SHA-256, highlighting a logic flaw when using hashes for authentication in a naïve way en.wikipedia.org . You successfully forged a MAC by extending the message, underscoring the need for constructions like HMAC en.wikipedia.org .

Through hands-on use of HashClash and HashPump, you gained practical experience with these cryptographic attacks. As an undergraduate student of information security, you should now better understand:

- ♦ The importance of collision resistance and the real impact when it's broken (how two different files can have the same "fingerprint" and the havoc that can cause).
- ♦ The inner workings of hash functions that allow length extension, and why certain uses of hashes are unsafe despite the hash function itself (SHA-256) being cryptographically strong against collisions or preimages.

Always keep in mind: when using cryptographic hashes in any security design, consider known vulnerabilities and use the hash in approved ways. If integrity and authenticity are needed, prefer HMAC or digital signatures over raw hashes. If using a hashing algorithm for storing passwords, use salt and proper schemes, etc. Cryptography is subtle – these labs showed just a couple of classic pitfalls. Being aware of them makes you a more secure developer and analyst.

Citations



Computer Security

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>



Computer Security

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>



Computer Security

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>



Computer Security

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>



Computer Security

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>



GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...

<https://github.com/cr-marcstevens/hashclash>

 **GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...**

<https://github.com/cr-marcstevens/hashclash>

 **GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...**

<https://github.com/cr-marcstevens/hashclash>

 **GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...**

<https://github.com/cr-marcstevens/hashclash>

 **GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...**

<https://github.com/cr-marcstevens/hashclash>

 **GitHub - cr-marcstevens/hashclash: Project HashClash - MD5 & SHA-1 cryptanal...**

<https://github.com/cr-marcstevens/hashclash>

 **Computer Security**

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>

 **Computer Security**

<https://www.cs.montana.edu/pearsall/classes/fall2023/476/labs/Lab9.html>

 **Length extension attack - Wikipedia**

https://en.wikipedia.org/wiki/Length_extension_attack

 **Length extension attack - Wikipedia**

https://en.wikipedia.org/wiki/Length_extension_attack

 **Length extension attack - Wikipedia**

https://en.wikipedia.org/wiki/Length_extension_attack

 **Length Extension Attack Demonstration – Dan Q**

<https://danq.me/2023/11/30/length-extension-attack/>

 **Length Extension Attack Demonstration – Dan Q**


<https://danq.me/2023/11/30/length-extension-attack/>

 **Length extension attack - Wikipedia**


https://en.wikipedia.org/wiki/Length_extension_attack

 **Length extension attack - Wikipedia**


https://en.wikipedia.org/wiki/Length_extension_attack

 **sTorro/HashPumpWIN32: A tool to exploit the hash length ... - GitHub**


<https://github.com/sTorro/HashPumpWIN32>

 **GitHub - miekrr/HashPump: A tool to exploit the hash length extension attack i...**

<https://github.com/miekrr/HashPump>

 **GitHub - miekrr/HashPump: A tool to exploit the hash length extension attack i...**

<https://github.com/miekrr/HashPump>

 **Introduction to hash length extension attacks and how to use Hashpump install...**


https://topic.alibabacloud.com/a/introduction-to-font-coloredhashfont-length-extension-attacks-and-how-to-use-hashpump-installation_8_8_31234758.html

 **Introduction to hash length extension attacks and how to use Hashpump install...**

https://topic.alibabacloud.com/a/introduction-to-font-coloredhashfont-length-extension-attacks-and-how-to-use-hashpump-installation_8_8_31234758.html

 **Introduction to hash length extension attacks and how to use Hashpump install...**

https://topic.alibabacloud.com/a/introduction-to-font-coloredhashfont-length-extension-attacks-and-how-to-use-hashpump-installation_8_8_31234758.html

 **GitHub - miekrr/HashPump: A tool to exploit the hash length extension attack i...**

<https://github.com/miekrr/HashPump>

 **Length Extension Attack Demonstration – Dan Q**


<https://danq.me/2023/11/30/length-extension-attack/>

 **Length Extension Attack Demonstration – Dan Q**

<https://danq.me/2023/11/30/length-extension-attack/>

All Sources

 [cs.montana](https://cs.montana.edu/)

 [github](https://github.com/)

 [en.wikipedia](https://en.wikipedia.org/)

 [danq](https://danq.me/)

 [topic.alibabacloud](https://topic.alibabacloud.com/)