Back to Course

Project 3

PYTHON GROUP

Details | Exploit | Fix

Exploit Details

Introduction to DNS Spoofing

- About
- How It Happens
- How to Prevent It
- Real Example
- Conclusion

About

When browsing the internet, the Domain Name System (DNS) plays an important role connecting users to websites. DNS translates domain names, like google.com, to an IP address that computers use to find each other on the web. This makes it possible for people to visit websites with simple, memorable names instead of having to remember a series of numbers. However, despite its convenience, DNS has some vulnerabilities. One significant threat is an attack known as DNS spoofing.

What is DNS Spoofing? DNS spoofing, also known as DNS cache poisoning, is a type of attack that modifies DNS records to redirect traffic to malicious sites or services. This allows attackers to intercept, alter, or steal data by making users believe they are interacting with legitimate sites or services.

How does DNS Spoofing Work? DNS spoofing happens when an attacker exploits vulnerabilities to insert fake information into the DNS cache. For example, when a user tries to visit a website, their browser requests the DNS server to convert the domain name to an IP address. If an attacker intercepts this request, they can respond with a fake IP address leading to a malicious site. The DNS server then stores this information, so future requests for that domain are redirected to the malicious site.

How It Happens

DNS spoofing can be done using several techniques. Here, we will go over two methods used by attackers to perform this attack. The code provided are simplified and are for demonstrative purposes.

Method 1: Flood Server with Fake DNS Responses This method targets the DNS server by overwhelming it with a large number of DNS responses. The attacker's goal is to trick the server into caching one of these responses, where each response is made to look like one that the server would normally process. Here's a code snippet illustrating how an attacker might flood a DNS server:

```
def send_fake_dns_response(domain, fake_ip, target_address):
  # Create a UDP socket
  sock = socket.socket(socket.AF_INET, socket.SOCK.DGRAM)

  # Create a fake DNS response
  # - add header
  # - add question
  # - add answer
  dns_response = ...

  # Send the packet multiple times to increase the chance of DNS spoofing
  for _ in range(1000):
```

```
    sock.sendto(dns_response, (target_address))

# Example usage
send_fake_dns_response("example.com", "fake_ip", ("target_ip", "target_port"))
```

Method 2: Man-in-the-Middle Attack In a Man-in-the-Middle (MitM) attack, the attacker is between the client and DNS server. By intercepting DNS queries, the attacker can alter the responses to redirect the client to malicious sites or services. To put themselves between the client and DNS server, attackers often use techniques like ARP spoofing, which tricks the client into sending their DNS queries to the attacker.

Simulating a Man-in-the-Middle Attack on Localhost: For this project, we can simulate a MitM attack by running an attacker DNS program in place of the local DNS. This setup allows us to see how the client program handles DNS responses from a malicious source.

```
class AttackerDNS:
  def __init__(self, fake_ip, address):
    self.fake_ip = fake_ip
    self.address = address
    self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.udp_socket.bind(address)

  def intercept_and_spoof(self):
    while True:
      # Receive DNS query from the client
      dns_query, client_address = self.udp_socket.recvfrom(4096)

      # Create a fake DNS response
      dns_response = self.create_fake_response(dns_query)

      # Send the fake response back to the client
      self.udp_socket.sendto(dns_response, client_address)

  def create_fake_response(self, dns_query):
    dns_response = ...
    return dns_response

# Example usage
attacker_dns = AttackerDNS("fake_ip", ("127.0.0.1", 21000))
attacker_dns.intercept_and_spoof()
```

How to Prevent It

To protect against DNS spoofing, we can implement several security measures. These strategies can significantly reduce the risk of spoofing, ensuring a safer user experience and maintaining the integrity of the DNS. Here are some key strategies to enhance DNS security:

Monitor and Detect Anomalies: Implement monitoring and detection systems to watch for suspicious activity. This could be unexpected DNS queries or responses that can indicate potential spoofing attempts.

Implement DNS Security Extensions (DNSSEC) DNSSEC adds a layer of security to the DNS protocol by using digital signatures to verify DNS responses. This helps ensure that the data received is authentic and has not been tampered with. In real DNS, this process involves many parts and entities. However, to understand the core concept, we can use a shared secret key between the client and DNS server.

Client: The client sends queries to the DNS server and verifies the responses using the shared key.

```
class DNSClient:
  def __init__(self, secret_key, address):
    self.secret_key = secret_key
    self.address = address
    self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

  def send_query(self, domain, dns_type):
    pass
```

```
  def create_query(self):
    pass

  def verify_signature(self):
    pass

# Example usage
dns_client = DNSClient("shared_secret_key", ("127.0.0.1", 21000))
dns_client.send_query("www.csusm.edu", "A")
```

DNS Server: The server handles incoming queries and sends signed responses using the shared key.

```
class LocalDNS:
  def __init__(self, secret_key, address):
    self.secret_key = secret_key
    self.address = address
    self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.udp_socket.bind(address)

  def handle_query(self):
    pass

  def send_response(self):
    pass

  def create_signed_response(self):
    pass

  def verify_signature(self):
    pass

# Example usage
local_dns = LocalDNS("shared_secret_key", ("127.0.0.1", 21000))
local_dns.handle_query()
```

Real Example

CVE-2023-41045 Understanding and Mitigating a DNS Spoofing Vulnerability

The Graylog log management platform had a critical vulnerability identified as CVE-2023-41045. This vulnerability was present in all versions up to and including 5.1.3 and allowed for DNS spoofing because of insecure source port usage for DNS queries. This example will show how DNS spoofing appears in real code and how the issue was fixed.

Understanding the Vulnerability: The core issue was that Graylog used only one source port for DNS queries. This went against recommended practices to make it hard for attackers to find the source port used for DNS queries. By binding a single socket for outgoing DNS queries and not changing the port number, Graylog increased the risk of DNS spoofing.

Vulnerable Code: In this code snippet, DnsClient only considered the query and request timeouts with no method to distribute the queries across a pool of sockets. This approach meant that the same socket was used for all DNS queries, making it vulnerable to DNS spoofing attacks.

```
public class DnsClient {
  ...
  public DnsClient(long queryTimeout, long requestTimeout) {
    this.queryTimeout = queryTimeout;
    this.requestTimeout = requestTimeout;
  }
  ...
}
```

Fixing the Vulnerability: To address this vulnerability, the developers implemented a change to distribute DNS queries through a pool of distinct sockets, each with a random source port. This approach maximizes the uncertainty in the choice of the source port for a DNS query, reducing the

risk of DNS spoofing.

```
public class DnsClient {
  ...
  public DnsClient(long queryTimeout, int resolverPoolSize, long resolverPoolRefreshSeconds) {
      this(queryTimeout, queryTimeout + DEFAULT_REQUEST_TIMEOUT_INCREMENT, resolverPoolSize, resolverPoolRefreshS
  }
  ...
}
```

By understanding and addressing the vulnerabilities in the DNS query process, the developers of Graylog were able to reduce the risk of DNS spoofing. This real example demonstrates the importance of securing DNS queries to prevent spoofing attacks.

Conclusion

DNS spoofing is a significant threat that can redirect users to fake sites or services, allowing attackers to intercept, alter, or steal data. By understanding how DNS spoofing works and the methods attackers use, we can implement effective security measures to protect against these threats.

Introduction to Deserialization of Untrusted Data

About

In software development, applications often need to exchange data. This requires converting complex data structures into a format that can be easily shared or stored. When an application receives this data, it should be able to convert it back to its original structure. This is a process known as serialization and deserialization.

What is Serialization? Serialization is the process of converting an object into a format that can be easily shared or stored. This format is usually a string or binary data. The following Python code demonstrates how serialization works using the `json` module:

```
# Serialization
user_info = {"name", "Alice", "age": 30}
serialized_json_string = json.dumps(user_info)
```

What is Deserialization? Deserialization is the reverse process, where the serialized data is converted back to its original object. The following Python code demonstrates how deserialization works:

```
# Deserialization
user_info = json.loads(serialized_json_string)
```

While serialization and deserialization are common practices in software development, it can introduce significant security risks when it comes to untrusted data. If an application deserializes data from unknown sources, it can open the door to many types of exploits.

How It Happens

Deserialization vulnerabilities happen when a program deserializes untrusted data without proper validation, making it open to attacks. Here are two common methods attackers use to exploit this vulnerability:

Method 1: Remote Code Execution (RCE) This method involves creating serialized data designed to execute arbitrary code on the target system. This attack takes advantage of the deserialization process when a program uses an unsafe deserialization library. This is a library that does not restrict which classes and methods can be deserialized. Here's an example illustrating how an attacker might exploit a program using the unsafe deserialization library `pickle`:

Server: The server deserializes incoming data.

```
def deserialize_data(data):
    return pickle.loads(data)

# Data from external source
data = sock.recvfrom(4096)

result = deserialize_data(data)
```

Attacker: The attacker creates malicious serialized data that will execute the `ls` command on the server. This is done using the `__reduce__` method, a special function that gets called during the deserialization process by `pickle`. Anything inside the `__reduce__` method will be executed.

```
class Message:
    def __init__(self, username, content):
        self.username = username
        self.content = content

    def __reduce__(self):
        return (os.system, ("ls",))

message = Message("user1", "Hello World!")

# Serialize data
malicious_data = pickle.dumps(message)

sock.sendto(malicious_data, ("target_ip", "target_port"))
```

How to Prevent It

To protect against deserialization vulnerabilities, we can implement several security measures. These strategies can significantly reduce the risks with deserialization. Here are some key strategies to prevent deserialization attacks.

Use Secure Serialization Formats Opt for deserialization libraries that enforce strict security measures. Avoid using unsafe libraries like `pickle` and using functions like `eval` in Python, which can be exploited by attackers. Both can execute arbitrary code, leading to severe security vulnerabilities. Instead, consider using safer alternatives like `json` or implement security measures that make deserialization of untrusted data safer.

Implement Data Validation Validate all incoming data before deserialization to ensure the data conforms to the expected format and structure. This helps prevent malicious inputs from being processed. Here's an example using regex to check if the data received from the socket includes the `__reduce__` method, which can run arbitrary code when using `pickle`:

```
def validate_data(data):
    if re.search(r"__reduce__", data):
        return False
    return True

def deserialize_data(data):
    return pickle.loads(data)

# Data from external source
data = sock.recvfrom(4096)

if validate_data(data):
    result = deserialize_data(data)
else:
    print("Potential malicious data detected!")
```

Real Example

CVE-2024-8514 Understanding and Mitigating a Deserialization Vulnerability

The Prisna GWT - Google Website Translator plugin for WordPress had a critical vulnerability identified as CVE-2024-8514. This vulnerability was present in all versions up to and including 1.4.11 and allowed for PHP Object Injection via deserialization of untrusted data. This example will show how deserialization of untrusted data appears in real code and how the issue was fixed.

Understanding the Vulnerability: The core issue was that the plugin deserialized untrusted data without proper validation. Specifically, the `prisna_import` parameter allowed an authenticated attacker to inject a PHP object, which allowed an attacker to execute arbitrary code.

Vulnerable Code: In this line, the `unserialize` function is used to convert the encoded data back into a PHP object. However, this function does not validate the data it processes. An attacker could create a serialized object that executes malicious code when it is deserialized.

```
$unserialize = @unserialize($decode);
```

Fixing the Vulnerability: To address this vulnerability, the developers added a validation step before deserialization. This step uses a regular expression to check for patterns observed in malicious serialized data. If such patterns are detected, the serialized data is replaced with an empty string.

```
$unserialize = preg_match('/O:\d+:(["\'])[^\1]+?\1:\d+:{/i', $decode) ? '' : @unserialize($decode);
```

By understanding and addressing the specific vulnerabilities in the deserialization process, the developers of the Prisna GWT plugin were able to prevent arbitrary code execution. This real example demonstrates the importance of validating untrusted data before deserialization.

Conclusion

Deserialization is a common practice in software development. Through careful attention to deserialization practices and implementing security measures, it is possible to protect applications from the dangers of untrusted data. This not only ensures the reliability and integrity of the software, but also protects users and their data from potential attacks.

Submit Exploit

Code to Exploit

Group 8

Download Code

Exploit type

Command Exploit Report