Michał Sajdak

Podstawy protokołu HTTP

WSTĘP

Nagłówki HTTP, URL, URI, żądania, odpowiedzi, kodowanie procentowe, formularze HTML, parametry przesyłane protokołem HTTP, różne implementacje serwerów HTTP skutkujące problemami bezpieczeństwa – to tylko kilka elementów, którymi zajmę się w tym tekście. Początkujący Czytelnicy poznają podstawy koniecznie do dalszego zgłębiania tematyki bezpieczeństwa aplikacji webowych, a nieco bardziej zaawansowani będą mieć okazję do spokojnego uporządkowania wiedzy.

W tekście skupię się na podstawach dotyczących protokołu HTTP w wersji pierwszej (dostępna jest też wersja druga[1], trwają prace nad trzecią[2]). Przedstawione tu informacje kierowane są przede wszystkim do osób zainteresowanych tematyką bezpieczeństwa aplikacji WWW. Nie jest to kompendium dotyczące protokołu HTTP. Zainteresowanych encyklopedyczną wiedzą odsyłam do dokumentów RFC opisujących HTTP w wersji 1.1[3]. Dla jasności przekazu dodam, że nie wspominam o elementach związanych z HTTPS.

Formalne specyfikacje, utarte przyzwyczajenia oraz rzeczywistość to trzy różne, często odległe sprawy. Wiele problemów bezpieczeństwa wynika właśnie z nieoczywistych różnic pomiędzy tymi trzema obszarami. Przykłady? Przejdźmy do konkretów dotyczących protokołu HTTP.

PODSTAWOWA KOMUNIKACJA HTTP

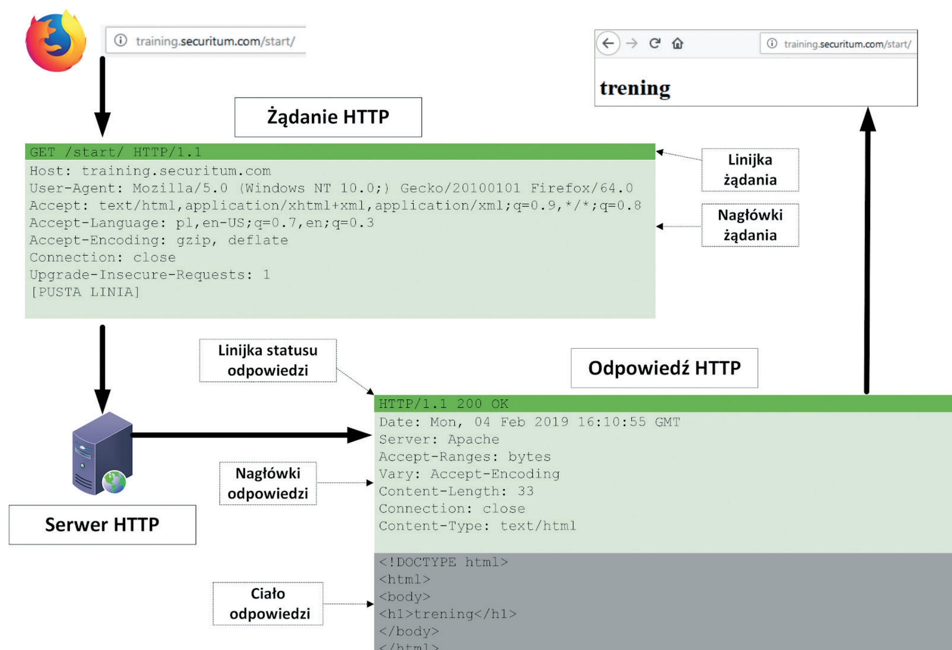
Najczęściej spotkamy się z komunikacją HTTP odbywającą się z wykorzystaniem protokołu TCP (choć czasem można spotkać wykorzystanie protokołu UDP[4]).

Komunikacja HTTP realizowana jest poprzez wysłanie żądania (ang. *request*) do serwera, który następnie generuje odpowiedź (ang. *response*). Przykład tego typu można zobaczyć na rysunku 1.

Analizę komunikacji zacznijmy od żądania HTTP (dla uproszczenia przykładu usunąłem z niego niewymagane nagłówki):

Listing 1. Prosta komunikacja HTTP

```
GET /start/ HTTP/1.1
Host: training.securitum.com
[pusta-linia]
```



Rysunek 1. Komunikacja HTTP

Pierwsza linijka żądania zawiera:

» metodę (ang. *method* lub *verb*; w naszym przypadku to: `GET`),

» adres URL (w naszym przypadku to: `/start/`),

» wersję protokołu (w naszym przypadku to: `HTTP/1.1`).

Zwracam uwagę na spacje, które rozdzielają powyższe elementy: musi być ich dokładnie dwie – zgodnie ze strukturą: `METODA[spacja]URL[spacja]WERSJA_HTTP`.

Druga linijka powyższego żądania zawiera nagłówek. Nazwa nagłówka to: `Host`, a jego wartość: `training.securitum.com`.

W formie bardziej niskopoziomowej komunikacja ta wygląda w następujący sposób:

```
▶ Internet Protocol Version 4, Src: 192.168.1.11, Dst: 45.56.85.138
▶ Transmission Control Protocol, Src Port: 51550, Dst Port: 80, Seq: 1, Ack: 1, Len: 54
▶ Hypertext Transfer Protocol

0000                                           45 00   .*....x0 C.5...E.
0010  00 6a 74 1e 40 00 40 06  81 fa c0 a8 01 0b 2d 38   .jt.@.@. ......-8
0020  55 8a c9 5e 00 50 e2 4f  40 e9 b8 2d 7d 75 80 18   U..^.P.O @..-}u..
0030  10 15 bb ea 00 00 01 01  08 0a 37 00 88 00 df 56   ........ ..7....V
0040  74 87 47 45 54 20 2f 73  74 61 72 74 2f 20 48 54   t.GET /s tart/ HT
0050  54 50 2f 31 2e 31 0d 0a  48 6f 73 74 3a 20 74 72   TP/1.1.. Host: tr
0060  61 69 6e 69 6e 67 2e 73  65 63 75 72 69 74 75 6d   aining.s ecuritum
0070  2e 63 6f 6d 0d 0a 0d 0a                           .com....
```

Rysunek 2. Komunikacja HTTP, warstwa Ethernet została celowo pominięta

W tym miejscu warto zwrócić uwagę na ostatnie cztery (zapisane szesnastkowo) znaki: `0d0a0d0a`. Ciąg: `0d0a` to separator linii w protokole HTTP. Często ciąg ten

oznaczany jest jako CRLF. Na marginesie dodam, że po polsku ktoś zaproponował całkiem poetyckie tłumaczenie nieco suchego CRLF[5]:

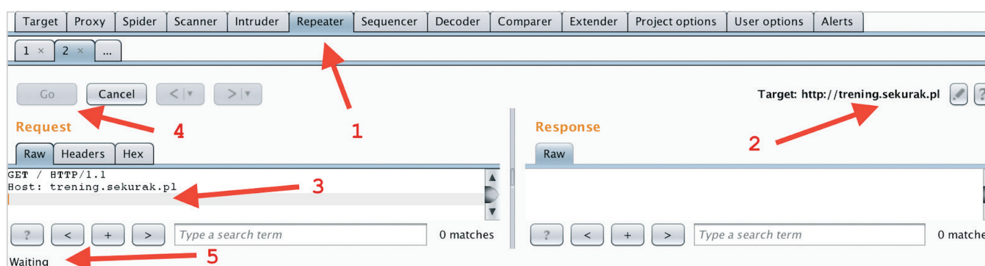
> *Ja bym poszedł bardziej w onomatopeję: KSZSZDING! (Zrozumieją tylko starzy, co mieli maszynę do pisania).*

Mamy więc jeden KSZSZDING! po nagłówku `Host` oraz drugi po wszystkich nagłówkach (wymaga tego od nas protokół HTTP). Po co skupiam się na takim szczególe? Ponieważ zdarza się wysyłanie żądań HTTP tylko z jednym znakiem końca linii, znajdującym się po nagłówkach, co nie jest poprawne i kończy się zazwyczaj oczekiwaniem serwera HTTP właśnie na znak CRLF (wygląda to tak, jakby serwer nie odpowiadał…).

ĆWICZENIE

Warto w tym miejscu wykonać proste ćwiczenie. Korzystając z bezpłatnej wersji narzędzia Burp Suite*, wyślij żądanie HTTP, kończąc je tylko jednym złamaniem linii. Przeanalizuj wynik, zanim zaczniesz czytać dalej.

Po wysłaniu takiej komunikacji nie dostajemy odpowiedzi:



Rysunek 3. Niedokończona komunikacja HTTP

W przypadku dwóch znaków końca linii wszystko pójdzie gładko:

Rysunek 4. Poprawna komunikacja HTTP

Dociekliwi Czytelnicy pewnie zauważą, że na żądaniu widocznym na rysunku 4 widać dwie puste linie (nie jedną, jak napisałem wcześniej). To jak w końcu ma to wyglądać? Jeszcze raz przypomnę – specyfikacja HTTP mówi o wymaganych dwóch

* Zob. rozdz. *Burp Suite Community Edition – wprowadzenie do obsługi proxy HTTP*.

ciągach `0d0a` (CRLF tudzież swojski KSZSZDING) po ostatnim nagłówku. Część osób interpretuje to jako jedną pustą linię, ale w niektórych przypadkach (Burp Suite) widzimy dwie. Obie interpretacje są poprawne, ważne tylko, żeby realna komunikacja wyglądała jak na rysunku 4 (mam na myśli sam koniec żądania). Po wysłaniu żądania serwer odpowiada nam w taki sposób:

Listing 2. Odpowiedź serwera na wysłane żądanie

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:15:03 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>trening</h1>
</body>
```

Widzimy tutaj następujące elementy:

» linijka statusu odpowiedzi: `HTTP/1.1 200 OK`,
» kolejne nagłówki odpowiedzi, np.: `Server: Apache`,
» pusta linijka,
» ciało (ang. *body*) odpowiedzi:

```
<body>
<h1>trening</h1>
</body>
```

Wróćmy do żądania HTTP opisanego w listingu 1 i omówmy nieco dokładniej jego elementy.

METODY HTTP

Tym razem spróbujemy użyć innych metod niż `GET`. Specyfikacja HTTP definiuje ich kilka[6], a tak naprawdę można spotkać ich nawet kilkadziesiąt[7]. Na początek spróbujmy wysłać żądanie metodą `HEAD`:

Listing 3. Żądanie wysłane metodą HEAD

```
HEAD / HTTP/1.1
Host: training.securitum.com
[pusta-linia]
```

Dostaniemy analogiczną odpowiedź jak wcześniej, ale już bez ciała odpowiedzi (jak widać, poprawna odpowiedź HTTP nie musi go zawierać), choć z jedną pustą linijką na końcu:

Listing 4. Odpowiedź HTTP na analizowane żądanie

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:37:40 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html
[pusta-linia]*
```

Gdzie taka metoda może się przydać? Choćby podczas próby siłowego lokalizowania pewnych ukrytych plików i katalogów:

```
HEAD /test HTTP/1.1
Host: training.securitum.com
```

W odpowiedzi uzyskujemy informację bez ciała odpowiedzi – dzięki temu przyspieszamy pojedynczy test (serwer musi wysłać mniej bajtów w odpowiedzi):

Listing 5. Odpowiedź serwera na żądanie

```
HTTP/1.1 404 Not Found
Date: Mon, 28 Jan 2019 18:39:18 GMT
Server: Apache
Vary: Accept-Encoding
Content-Type: text/html; charset=iso-8859-1
```

Warto zauważyć, że zasób `/` istnieje (otrzymaliśmy kod odpowiedzi `200` – por. listing 4), zasób `/test` nie istnieje (otrzymaliśmy kod błędu `404` – por. listing 5). Oczywiście nie zawsze tak musi być – serwer przykładowo może zwrócić kod `200`, a o problemie poinformować nas w ciele odpowiedzi.

Czy są dostępne inne metody? Tak – np. `OPTIONS` wskazująca obsługiwane przez serwer metody (warto pamiętać, że informacja zwracana w odpowiedzi nie musi być zawsze prawdziwa):

* Dla uproszczenia w kolejnych przykładach będę pomijał oznaczenie `[pusta-linia]`.

Listing 6. Inne metody HTTP

```
OPTIONS / HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 18:41:41 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html
```

W kontekście naszych rozważań szczególnie niebezpieczna bywa metoda `PUT` (często jest ona domyślnie wyłączona), umożliwiająca tworzenie plików na serwerze HTTP:

Listing 7. Przykład użycia metody `PUT`

```
PUT /test2.php HTTP/1.1
Host: training.securitum.com
Content-Length: 19

<?php phpinfo(); ?>
```

```
HTTP/1.1 201 Created
Location: /test2.php
```

Jak widzimy, w ciele żądania podajemy zawartość pliku, a w pierwszej linijce – jego nazwę. Jeśli metoda jest obsługiwana, moglibyśmy teraz wysłać żądanie do nowo utworzonego pliku `test2.php` – i w zasadzie mamy możliwość wykonania dowolnego kodu na serwerze! Przy okazji warto też dodać, że metody nie zawsze muszą być zapisywane wielkimi literami. Być może metoda `PUT` jest zabroniona, ale `pUt` już nie!

Listing 8. Użycie metody `pUt`

```
pUt /test2.php HTTP/1.1
Host: training.securitum.com
Content-Length: 19

<?php phpinfo(); ?>
```

W tym miejscu warto też wspomnieć, że po ciele zapytania nie mamy już złamania linii (CRLF). Może ono tam występować, ale będzie traktowane jako zawartość ciała żądania.

URL CZY URI?

Na wstępie zaznaczę, że w kontekście HTTP bardzo popularnym terminem jest URL (*Uniform Resource Locator*). Historycznie[8] był to po prostu adres, którego można użyć w HTML-owym hiperlinku, np.: *http://training.securitum.com/request.php?param=wartosc*.

Często „adres URL" to po prostu ciąg, który wpisujemy w pasek adresu (nazywany czasem wprost paskiem URL) przeglądarki. Wygląda prosto oraz intuicyjnie, prawda? Problem w tym, że w wielu specyfikacjach[9] nie znajdziemy pojęcia URL. Często mowa będzie jedynie o URI (*Uniform Resource Identifier*). Formalnie rzecz ujmując, URL-e są podzbiorem zbioru URI[10]. Czy powinniśmy więc wymazać z pamięci „herezję URL-i" i posługiwać się pojęciem URI? Zdecydowanie nie. Wiele osób używa tych terminów zamiennie (bez straty dla precyzji rozumowania). Powstał nawet stosowny dokument RFC wyjaśniający całe zamieszanie[11]. W każdym razie na nasze potrzeby będziemy używali zamiennie obu terminów: URL oraz URI.

Bardzo często można spotkać się z tego typu URL-ami: *http://training.securitum.pl/katalog/test.php?parametr=wartosc#fragment*.

Możemy je jednak rozszerzyć (zauważmy zmieniony port oraz opcjonalną część dotyczącą użytkownika i hasła): *http://uzytkownik:hasło@training.securitum.pl:8042/katalog/test.php?parametr=wartosc#fragment*.

Zwróćmy uwagę na `fragment`. Jest to popularne odwołanie do kotwicy w HTML, a wpisanie takiego URL-a w przeglądarkę nie wysyła do serwera znaku # oraz tego, co znajduje się po nim. Co ciekawe, użycie URL-a z użytkownikiem i hasłem bezpośrednio w żądaniu HTTP nie jest poprawne:

```
GET http://login:haslo@training.securitum.com/ HTTP/1.1
Host: training.securitum.com
```

Czy zatem zawsze w odpowiedzi otrzymamy błąd `HTTP/1.1 400 Bad Request`? Często tak, choć podejrzewam, że mogą znaleźć się serwery HTTP, które bez problemu pozwolą na tego typu URI. URL-e mogą też mieć formę względną, np.: */katalog/test.php?parametr=wartosc*. I to właśnie ją najczęściej widzimy w żądaniach HTTP.

Uzbrojeni w podstawy teoretyczne dotyczące pojęcia URL (czy też URI), zobaczmy prosty przykład żądania:

```
GET /test HTTP/1.1
Host: training.securitum.com
```

W tym przypadku URL jest względny: `/test` jednak śmiało możemy tu również zastosować w wersji bezwzględnej:

Listing 9. Przykład żądania i odpowiedzi HTTP z bezwzględnym URL

```
GET http://training.securitum.com/test HTTP/1.1
Host: trening2.sekurak.pl
```

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 19:29:28 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 18:11:05 GMT
ETag: "2d2e0-21-5808899aa4c5d"
Accept-Ranges: bytes
Content-Length: 33
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>trening</h1>
</body>
```

Rzadko można znaleźć tego typu URL w rzeczywistych żądaniach HTTP, ale jest to jak najbardziej poprawne i może będzie miało jakieś ciekawe implikacje związane z bezpieczeństwem? W powyższym przykładzie wartość nagłówka `Host` zostanie najczęściej zignorowana przez serwer. Tego typu zachowanie może być czasem wykorzystane do zlokalizowania podatności[12].

> *Niektóre serwery HTTP whitelistują nagłówek host, ale zapominają, że w linijce żądania również znajdować się może adres serwera [host]* i to właśnie on ma pierwszeństwo**.*

Listing 10. Przykład adresu serwera w linijce żądania

```
GET http://internal-website.mil/ HTTP/1.1
Host: xxxxxxx.mil
Connection: close
```

Czyli z jednej strony po stronie serwerowej sprawdzane jest, czy nagłówek `Host` przyjmuje jedną z dozwolonych wartości (żądanie jest więc dopuszczone do dalszego przetwarzania), z drugiej strony, ktoś zapomniał, że pierwszeństwo ma URL z pierwszej linijki (więc *de facto* otrzymamy dostęp do tego właśnie zasobu)!

Warto tutaj zwrócić uwagę na jeszcze jeden ciekawy fakt. URL jest często normalizowany przez przeglądarkę, np. wpisanie w pasek przeglądarki adresu: *training.securitum.com/test/../* powoduje wysłanie do serwera wersji znormalizowanej, czyli:

```
GET / HTTP/1.1
Host: training.securitum.com
```

* Czyli pozwalają wykonać żądanie tylko wtedy, jeśli nagłówek `host` posiada jedną z dozwolonych w konfiguracji wartości.

** „Some servers effectively whitelist the host header, but forget that the request line can also specify a host that takes precedence over the host header". Cyt. za: Kettle J., *Cracking the lens: targeting HTTP's hidden attack-surface, 2017*, https://portswigger.net/blog/cracking-the-lens-targeting-https-hidden-attack-surface [W całym rozdziale przekład własny Autora – przyp. red.].

Dlatego w naszych zastosowaniach do wysyłania żądań HTTP lepiej posługiwać się innym narzędziem niż przeglądarka – np. Burp Suite czy OWASP ZAP. Na koniec warto jeszcze wspomnieć o względnym URI następującego typu:

```
GET /../../../../../../../../../etc/passwd HTTP/1.1
Host: training.securitum.com
```

Popularne,nowoczesne serwery WWW (np. od Apache czy Microsoftu) nie pozwolą na takie żądanie. Jednak już np. w świecie IoT wszystko jest możliwe[13].

NAGŁÓWKI HTTP

Do tej pory skupiliśmy się głównie na pierwszej linijce żądania. Teraz popatrzmy na jedyny, wymagany w wersji 1.1 protokołu HTTP nagłówek `Host`. Czy podana tutaj nazwa to po prostu adres domenowy, który wpisujemy w przeglądarce? Bardzo często odpowiedź brzmi: tak, ale sprawdźmy, w jaki sposób przeglądarka łączy się z serwerem HTTP:

- w pasek przeglądarki wpisujemy: *training.securitum.com*,
- realizowane jest zapytanie do serwera DNS, dające nam adres IP: 178.32.219.59,
- przeglądarka łączy się z tym adresem z wykorzystaniem protokołu HTTP (na port 80) oraz wysyła wcześniej wspomniane żądanie HTTP z nagłówkiem: `Host: training.securitum.com`.

Czy możemy w wartości nagłówka `Host` użyć innej nazwy? Jak najbardziej – nawet takiej, której nie mamy w DNS:

Listing 11. Zmodyfikowane zapytanie i odpowiedź serwera

```
ptest$ host training2.securitum.com
Host trening2.securitum.com not found: 3(NXDOMAIN)
```

```
GET / HTTP/1.1

Host: training2.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 28 Jan 2019 19:02:38 GMT
Server: Apache
Last-Modified: Mon, 28 Jan 2019 19:02:04 GMT
ETag: "2d2e1-29-580894ff98fe3"
Accept-Ranges: bytes
Content-Length: 41
Vary: Accept-Encoding
Content-Type: text/html

<body>
<h1>to jakas staroc</h1>
</body>
```

Zauważmy, że ta metoda może służyć do lokalizowania ukrytych domen wirtualnych na danym serwerze, a to, że nie ma ich w DNS, w niczym nie przeszkadza*.

Autor cytowanego znaleziska przez manipulacje nagłówkiem `Host` uzyskał dostęp do domeny *yaqs.googleplex.com*, gdzie znajdowały się poufne informacje firmy Google. Za zgłoszenie podatności badacz otrzymał 10 tyś dolarów w ramach oficjalnego programu *bug bounty*.

Czy istnieją inne nagłówki HTTP? Oczywiście. Warto przy tym wspomnieć, że czymś innym są nagłówki przekazywane w żądaniu, a czym innym nagłówki wysyłane w odpowiedzi. Część nagłówków może nazywać się tak samo, np. `Content-Type`. W takich przypadkach warto dodać, czy chodzi nam o nagłówek żądania, czy odpowiedzi. Poniżej kilka nagłówków wartych wspomnienia:

» `Cookie` (nagłówek żądania) – wysyła ciasteczko lub ciasteczka do serwera,
» `Set-Cookie` (nagłówek odpowiedzi) – ustawia ciasteczko klientowi,
» `Server` (nagłówek odpowiedzi) – zdradzający czasem typ/wersję wykorzystywanego serwera HTTP,
» `X-Forwarded-For` (nagłówek żądania) – wyjątkowo ciekawy nagłówek z potencjałem naruszania bezpieczeństwa[14],
» `Strict-Transport-Security` (nagłówek odpowiedzi) – jeden z nagłówków mogących wprost zwiększyć bezpieczeństwo aplikacji[15],
» `Location` (nagłówek odpowiedzi) – realizuje przekierowanie klienta na inny adres. Przykład odpowiedzi z nagłówkiem `Location` poniżej:

Listing 12. Odpowiedź serwera z nagłówkiem `Location`

```
HTTP/1.1 302 Found
Date: Tue, 29 Jan 2019 17:16:35 GMT
Server: Apache
Location: http://sekurak.pl/
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html
```

Nieco więcej uwagi warto poświęcić nagłówkowi `Referer` (nagłówek żądania). Jego wartością jest adres URL strony poprzednio odwiedzanej przez użytkownika. Czyli jeśli kliknę np. w zewnętrzny link znajdujący się pod adresem: *https://sekurak.pl/teksty/?SID=19283312873872382* – moja przeglądarka wyśle do linkowanej strony nagłówek `Referer` z wartością będącą pełnym URL strony, na której przebywałem, czyli: *https://sekurak.pl/teksty/?SID=19283312873872382*. Dodatkowo web serwer w docelowym miejscu prawdopodobnie domyślnie ma włączone logowanie wartości nagłówka `Referer`. W efekcie z mojego serwisu mogą wyciekać pewne wrażliwe informacje (w tym przypadku może to być wartość zmiennej SID odpowiedzialnej za identyfikator sesji).

* Konkretny przykład tego typu działania można zobaczyć tutaj: Pereira E., *$10k host header*, https://www.ezequiel.tech/p/10k-host-header.html.

Ważna uwaga: mimo tego, że moja przeglądarka szyfruje ścieżkę w URL, jeśli używam protokołu HTTPS[16], to nagłówek `Referer` zostanie wysłany, jeśli linkowana strona również używa HTTPS. W ten sposób mogą wyciec dane przechowywane w URL-u – mimo korzystania z HTTPS.

Inny przykład, kiedy ten nagłówek może prowadzić do naruszania bezpieczeństwa, to użycie go w połączeniu z mechanizmem resetu hasła. W tym scenariuszu użytkownik sam inicjuje reset hasła, klikając w (poprawny) link w wiadomości e-mail. Następnie trafia np. na stronę: *https://sekurak.pl/reset_pass?token=JeHOkeh78f92Hzpo^2*.

Jeśli sekurak korzysta np. ze skryptów analitycznych (osadzonych w zewnętrznych domenach), to przeglądarka wyśle (w tle) stosowne żądania HTTP, z nagłówkiem `Referer` ustawionym na: *https://sekurak.pl/reset_pass?token=JeHOkeh78f92Hzpo^2*. Zatem zewnętrzna domena uzyskała dostęp do tokena umożliwiającego reset hasła. Pozostaje oczywiście pytanie, czy tokena można użyć tylko jeden raz oraz czy w łatwy sposób można rozpoznać użytkownika, którego hasło zostało zresetowane.

Aby przeglądarka nie wysyłała nagłówków `Referer`* z naszej domeny, można użyć nagłówka odpowiedzi: `Referrer-Policy: no-referrer`[17].

Podobnie jak w przypadku nagłówka `Referer`, na osobną uwagę zasługuje nagłówek `Host`. Wydaje się w zasadzie bezpieczny, choć wspomniałem wcześniej pewną ciekawostkę związaną z domeną *yaqs.googleplex.com*.

Nagłówek ten może być czasem użyty do wykorzystania podatności SSRF[18]. W tym przypadku, aby zmusić serwer do wykonania żądania HTTP do stosownego zasobu, wystarczyło użyć np. takiego polecenia:

```
curl -X POST -H 'Host: 162.243.147.21:81' 'https://gitlab.com/-/jira/login/
oauth/access_token'
```

Pamiętajmy też, że w wartości nagłówka `Host` mogą znajdować się również adresy IPv6, co czasem może prowadzić do ciekawych błędów[19].

Inny przykład złośliwego wykorzystania nagłówka `Host` można spotkać przy atakach na mechanizmy resetu hasła. Najczęściej wygląda to w ten sposób:

1. Atakujący (znający e-mail ofiary) inicjuje reset hasła – ale wchodzi na stronę z tym mechanizmem, wpisując nagłówek `Host: sekurak.pl`.
2. Podatna aplikacja używa nagłówka `Host` do wygenerowania linka z resetem hasła.
3. Taki link jest wysyłany (z prawidłowej domeny) do ofiary.
4. Jeśli ofiara kliknie w linka (nie musi resetować hasła) – to prawdopodobnie wyśle jednorazowy kod (znajdujący się w URL-u) na serwer atakującego.
5. Atakujący przejmuje dostęp do konta.

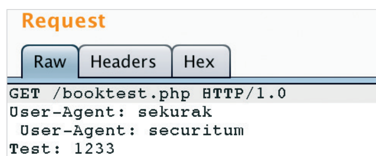
* Zwracam również uwagę na zapis nagłówka `Referer` – ma tylko trzy litery „r”, zaś `Referrer-Policy` – cztery.

Podatność ta została już wykorzystana wiele razy – warto zapoznać się z opisem tych ataków*.

Zbliżając się do końca rozważań o nagłówkach, warto jeszcze raz wspomnieć, że ich nazwy mogą być pisane dowolną kombinacją małych i dużych liter, np.: `hOSt`**.

ĆWICZENIE

Na koniec rozważań o nagłówkach zadanie dla dociekliwych. Jaka będzie wartość nagłówka `User-Agent` (po stronie serwerowej)? sekurak? securitum? Czy coś innego?



Rysunek 5. Jaką wartość ma `User-Agent` po stronie serwerowej?

Omawiany przykład to tzw. `folded header` (nagłówek z wartością zapisaną w kilku linijkach). Informacje o nim można znaleźć w stosownym dokumencie RFC:

> *Historycznie, wartość nagłówka HTTP mogła znajdować się w wielu następujących po sobie linijkach żądania. Każda kolejna linijka musi zaczynać się od przynajmniej jednej spacji lub znaku tabulatora****.*

Mimo że jest to przypadek historyczny, to składnia cały czas bywa obsługiwana – również w ten sposób, że generuje to problemy bezpieczeństwa****.

Czy nagłówki są absolutnie wymagane?

Chyba najczęściej spotykaną wersją protokołu HTTP jest 1.1. Dość powszechna jest jednak obsługa protokołu w wersji 1.0, kiedy w żądaniu nie musi występować żaden nagłówek:

---

* Zob. Cable J. (cablej), *Password reset link injection allows redirect to malicious URL*, https://hackerone.com/reports/281575; Cable J. (cablej), *Don't Trust the Host Header for Sending Password Reset Emails*, https://lightningsecurity.io/blog/host-header-injection/; Golunsky D., *CVE-2017-8295: WordPress 2.3-4.8.3 Unauthorized Password Reset/ Host Header Injection Vulnerability Exploit*, https://www.vulnspy.com/en-cve-2017-8295-unauthorized-password-reset-vulnerability/; Corben L. (cdl), *Password Reset link hijacking via Host Header Poisoning*, https://hackerone.com/reports/226659.

** Dla Czytelników dociekliwych i szukających źródeł informacji o nagłówkach polecam zapoznanie się z artykułami zamieszczonymi np. tutaj: *HTTP headers*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers, czy: *List of HTTP header fields* [w:] *Wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/List_of_HTTP_header_fields.

*** „Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold)"; por.: *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, https://tools.ietf.org/html/rfc7230.

**** Zob. np.: stbuehler, *Rewizja df8e4f95*, https://redmine.lighttpd.net/projects/lighttpd/repository/revisions/df8e4f95614e476276a55e34da2aa8b00b1148e9; Spek van der O., *Crash on duplicated headers with folding*, https://download.lighttpd.net/lighttpd/security/lighttpd_sa2007_03.txt; *Vulnerability Details: CVE-2017-5660*, https://www.cvedetails.com/cve/CVE-2017-5660/.

Listing 13. Przykład komunikacji protokołem HTTP w wersji 1.0 bez nagłówka w żądaniu

```
GET / HTTP/1.0
```

```
HTTP/1.1 200 OK
Date: Wed, 10 Apr 2019 13:46:18 GMT
Server: Apache
```

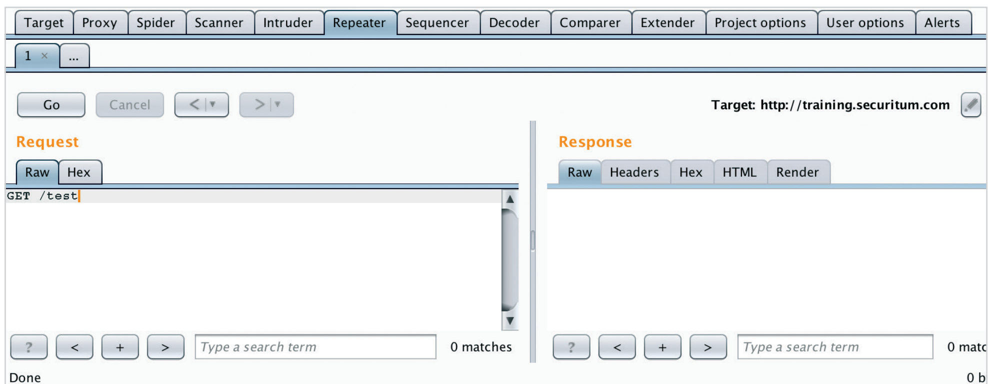
Czasem jest też obsługiwana zupełnie już archaiczna wersja protokołu 0.9, która w ogóle nie zna pojęcia nagłówka HTTP (ani w żądaniu, ani w odpowiedzi).

Listing 14. Przykład protokołu HTTP w wersji 0.9 – nie są tu widoczne nagłówki

```
GET /test
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /test was not found on this server.</p>
</body></html>
```

Jest to na tyle rzadko spotykana komunikacja, że nawet popularny Burp Suite (wersja 1.7.26 czy 2.1) nie obsługuje jej poprawnie (rysunek 6).



Rysunek 6. Komunikacja HTTP/0.9

W tym przypadku komunikacja HTTP jest rzeczywiście wysyłana do serwera, serwer zwraca odpowiedź, ale Burp nie wyświetla jej już poprawnie.

WARTOŚCI PRZEKAZYWANE DO APLIKACJI PROTOKOŁEM HTTP

To jeden z najistotniejszych, podstawowych tematów w kontekście bezpieczeństwa. Przytłaczająca liczba podatności w aplikacjach webowych może być wykorzystana

dzięki odpowiedniemu (złośliwemu) manipulowaniu wartościami przekazywanymi do aplikacji. W którym zatem miejscu żądania HTTP mogą znaleźć się parametry? Nieco wymijająca odpowiedź brzmi: wszędzie. Poczynając od linijki żądania, przez nagłówki, aż po ciało. Przyjrzyjmy się kilku popularnym miejscom, w których możemy znaleźć parametry.

Nagłówki żądania HTTP

To miejsce (może poza nagłówkiem `Cookie`, o którym będzie mowa nieco później) jest często niesłusznie ignorowane w kontekście potencjalnego zagrożenia związanego z niebezpiecznym użyciem wartości nagłówka. Ilustracją tego problemu będą dwie podatności. Pierwsza to *SQL injection*, zlokalizowana 22 marca 2019 roku w nagłówku `User-Agent`[20]. Nagłówek z wstrzyknięciem mógł w tym przypadku wyglądać np. tak:

```
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/55.0.2883.87'XOR(if(now()=sysdate(),sleep(5*5),0))OR'
```

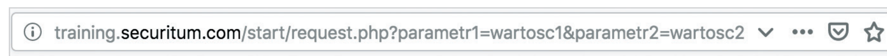
Druga podatność to *path traversal* w Ruby on Rails (CVE-2019-5418). W tym przypadku zupełnie niewinny nagłówek żądania `Accept` (mówiący normalnie, jakie formaty odpowiedzi akceptuje klient) mógł w pewnych sytuacjach doprowadzić do możliwości nieautoryzowanego odczytywania plików na serwerze:

```
Accept: ../../../../../../../../../../etc/passwd{{
```

Czytelnicy bardziej zainteresowani tym tematem z pewnością znajdą bogatą literaturę omawiającą szczegóły tej podatności[21].

URL

Parametry przekazywane w URL najczęściej zobaczymy w zapytaniach typu `GET`. Wielu z nas widziało na pewno tego typu konstrukcję, umieszczoną w pasku adresu przeglądarki internetowej:



Rysunek 7. Parametry przekazywane w pasku URL aplikacji

Przeglądarka na podstawie takiego adresu realizuje następujące żądanie HTTP (usunąłem z niego mniej istotne nagłówki):

Listing 15. Żądanie HTTP z parametrami w URL

```
GET /start/request.php?parametr1=wartosc1&parametr2=wartosc2 HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2019 09:15:30 GMT
Server: Apache
Content-Length: 39
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartosc2
```

Pamiętajmy, że to tylko pewna konwencja. Aplikacja jako wartości może równie dobrze użyć nazwy parametru (w naszym przypadku jest to `parametr1`). Co się stanie, gdy dodamy jakiś dodatkowy parametr, nieobsługiwany przez aplikację? Na przykład `parametr3`? Najczęściej – nic:

Listing 16. Przesłanie nieobsługiwanego przez aplikację parametru

```
GET /start/request.php?parametr1=wartosc1&parametr2=wartosc2&parametr3
=wartosc3 HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2019 09:16:28 GMT
Server: Apache
Content-Length: 39
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartosc2
```

Czasem jednak aplikacja analizuje wszystkie przesłane (np. metodą `GET`) parametry, więc warto sprawdzić, co się stanie, jeśli przekażemy tego typu dodatkową wartość.

Dociekliwi Czytelnicy mogą zapytać, czy da się przesłać w wartości parametru np. znak `&` (jak widać wyżej, ma on specjalne znaczenie) lub `#` (wcześniej wspomniałem, że znak # wpisany w pasek przeglądarki nie jest wysyłany do serwera). Oczywiście, można to zrobić, używając kodowania procentowego[22], nazywanego często kodowaniem URL. Działa ono w prosty sposób: kod ASCII znaku `&` to szesnastkowo `26`. W kodowaniu procentowym `%26`. Jeśli z kolei chcemy użyć spacji w URL, musimy ją zakodować jako `%20` (lub jako `+`). Z tego powodu użycie znaku „plus" też musi być zakodowane (jako `%2b`), w przeciwnym wypadku oznaczałoby zakodowaną spację. Czy możemy zakodować inne znaki (np. w nazwie parametru)? Jak najbardziej:

Listing 17. Wykorzystanie kodowania procentowego

```
GET /request%2ephp?parametr1=a%2bb&p%61rametr2=a%26+b%20c HTTP/1.1
Host: training.securitum.com
```

```
HTTP/1.1 200 OK
Date: Tue, 29 Jan 2019 19:47:52 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 32
Content-Type: text/html

parametr1: a+b
parametr2: a& b c
```

Zdarza się czasem, że kodowanie procentowe mylone jest z kodowaniem z wykorzystaniem encji, używanym często w HTML, np.: `<` `&`. To zupełnie różne kodowania i tego drugiego nie możemy użyć w URL-u.

Na koniec warto wspomnieć, że zaleca się, aby metodą `GET` nie przesyłać poufnych czy wrażliwych informacji, np. haseł czy identyfikatorów sesyjnych. Dlaczego? Parametry te widać bezpośrednio w pasku URL przeglądarki. Są one domyślnie logowane przez serwery HTTP, jak również widoczne w wynikach wyszukiwarek internetowych.

POST: application/x-www-form-urlencoded

Drugim często spotykanym sposobem wysyłania parametrów jest przesyłanie ich w ciele żądania, z wykorzystaniem metody `POST`. Często w ten sposób wysyłane są dane z formularzy HTML:

Listing 18. Formularz wysyłany metodą POST

```
<form action="/request.php" method="POST">
Podaj parametr 1: <input type="text" name="parametr1">
Podaj parametr 2: <input type="text" name="parametr2">
<input type="submit">
</form>
```

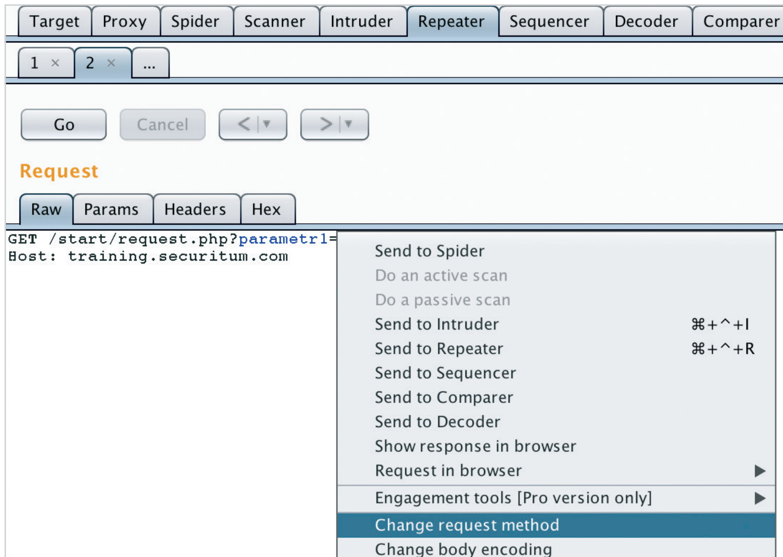
W tagu `<form>` możemy podać również parametr `enctype`, wskazujący przeglądarce, w jaki sposób powinna przesłać dane z formularza w żądaniu:

```
<form action="/request.php" method="POST"
enctype="application/x-www-form-urlencoded">
```

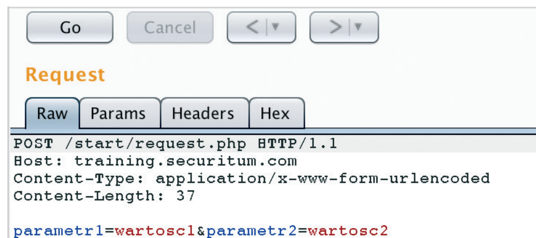
Wartość `application/x-www-form-urlencoded` jest domyślna (jeśli więc chcemy z niej skorzystać, możemy całkowicie pominąć parametr `enctype`), inne możliwości to `multipart/form-data` (więcej o niej w dalszej części tekstu) oraz `text/plain`.

ĆWICZENIE

W tym momencie proponuję zrealizować proste ćwiczenie, polegające na automatycznej zmianie metody `GET` na `POST`. W narzędziu Burp Suite (moduł REPEATER) wpisujemy żądanie z listingu 15, następnie klikamy prawym przyciskiem myszy i wybieramy opcję CHANGE REQUEST METHOD (patrz rysunek 8). Powinniśmy uzyskać efekt widoczny na rysunku 9.



Rysunek 8. Zmiana metody żądania HTTP



Rysunek 9. Żądanie HTTP typu `POST`

Czy zawsze zadziała zmiana metody `GET` na `POST` (lub na odwrót)? Niekoniecznie, choć dość często jest to możliwe. Czasem atakujący, uzyskujący nieautoryzowany dostęp do aplikacji, będą chcieli przekazywać parametry właśnie w ten sposób. Dlaczego? Parametry przesyłane metodą `POST` nie są domyślnie logowane przez serwery HTTP (w przeciwieństwie do parametrów przekazywanych w URL).

Wracając do szczegółów naszego nowego żądania, zauważmy, że poza zmianą metody pojawiły się dwa dodatkowe nagłówki:

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 43
```

Natomiast same parametry znalazły się w ciele żądania (następują po jednej pustej linijce). Uwaga – tutaj na końcu żądania nie mamy już znaków CRLF. Technicznie rzecz ujmując, mogą się tam znaleźć, ale wtedy będą częścią wartości parametru `parametr2` (zmieni się wtedy również `Content-Length` – wskazujący na długość ciała). Czy te same parametry mogą być przekazywane równocześnie w linijce żądania i w ciele? Jak najbardziej:

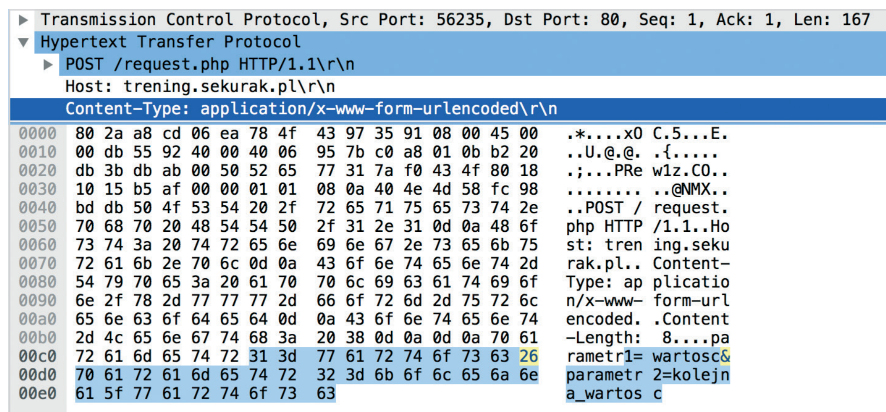
Listing 19. Parametry przekazywane w URL oraz w ciele żądania

```
POST /start/request.php?parametr1=wartosc1&parametr2=wartosc2 HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 37

parametr1=wartosc1&parametr2=wartosc2
```

Niekiedy prowadzi to do powstania interesujących podatności, takich jak np. luka w API WordPressa umożliwiająca anonimową zmianę dowolnego wpisu[23].

A jeśli zmienimy wartość nagłówka `Content-Length`? Czy zostanie wysłanych mniej danych? Nic z tego – wysłane zostanie pełne żądanie, a dopiero web serwer odpowiednio przetworzy całość (rysunek 10):



Rysunek 10. Wartość nagłówka `Content-Length` mniejsza niż długość ciała żądania – widok żądania HTTP wysłanego narzędziem Burp Suite

Listing 20. Wartość nagłówka `Content-Length` mniejsza niż długość ciała żądania

```
POST /start/request.php HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

parametr1=wartosc1&parametr2=wartosc2
```

Takie zachowanie może czasem posłużyć np. do omijania firewalli aplikacyjnych[24].

Niektórzy twierdzą, że ludzie zajmujący się bezpieczeństwem są czasem trochę złośliwi. Jest w tym chyba ziarno prawdy, bo jak nazwać wpisanie ujemnej wartości w nagłówek `Content-Length`? Jest to jak najbardziej możliwe, a efekty mogą być rozmaite – od błędów typu *Denial of Service*[25], aż po potencjał na wykonanie kodu na serwerze[26].

Przy okazji warto również wspomnieć o szalenie ciekawym opracowaniu *HTTP Desync Attacks: Request Smuggling Reborn* autorstwa Jamesa Kettle'a[27]. Kluczową rolę gra tutaj dość mało znany nagłówek `Transfer-Encoding`. Otóż można go użyć alternatywnie do omawianego wcześniej `Content-Length`. Przykład tego typu komunikacji widać w listingu 21. `2b` (liczba szesnastkowa) oznacza długość pierwszego „kawałka" (ang. *chunk*) ciała. Z kolei `0` z dwoma kszszdingami na końcu oznacza koniec ciała.

Listing 21. Użycie nagłówka `Transfer-Encoding` z wartością `chunked` – alternatywnie do nagłówka `Content-Length`

```
POST /start/request.php HTTP/1.1
Host: training.securitum.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

2b
parametr1=wartosc1&parametr2=wartoscwartosc
0
```

```
HTTP/1.1 200 OK
Date: Thu, 08 Aug 2019 19:19:17 GMT
Server: Apache
Content-Length: 45
Content-Type: text/html; charset=UTF-8

parametr1: wartosc1
parametr2: wartoscwartosc
```

Do czego ten mechanizm może zostać wykorzystany? Do nieoczekiwanych wstrzyknięć żądań HTTP, możliwych w przypadkach, kiedy w ich przetwarzaniu bierze udział wiele serwerów HTTP (np. serwer frontowy sprawdzający m.in. uprawnienia użytkownika do konkretnych URL-i oraz serwer backendowy realizujący logikę biznesową). Atak polega najczęściej na jednoczesnym użyciu w żądaniu dwóch nagłówków: `Content-Length` oraz `Transfer-Encoding` z wartością `chunked`. Dodatkowo, aby atak się powiódł, wymagane jest wysłanie dwóch lub więcej żądań – pierwsze „zatruwa" komunikację, kolejne realizują cel atakującego. Może brzmieć to nieco abstrakcyjnie, zobaczmy zatem konkretny, wybrany przykład.

Celem atakującego jest dostęp do katalogu `/admin`. Odpowiednie uprawnienia sprawdza serwer frontowy. W celu ominięcia zabezpieczeń w pierwszej kolejności realizowane jest tego typu żądanie:

Listing 22. Jednoczesne użycie nagłówków `Content-Length` oraz `Transfer-Encoding: chunked`

```
POST /home HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 62
Transfer-Encoding: chunked

0


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

W tym przypadku serwer frontowy widzi dostęp do zasobu `/home` oraz w celu określenia długości ciała żądania używa nagłówka `Content-Length` (zatem ostatnia linijka ciała żądania to `Foo: x`). Żądanie jest przesyłane dalej do serwera backendowego (dostęp do `/home` posiada każdy użytkownik), który w celu określenia długości ciała używa z kolei nagłówka `Transfer-Encoding`. W tym przypadku `0` (oraz podwójny CRLF) oznacza koniec ciała. Takie właśnie żądanie jest przetwarzane. Co jednak z trzema ostatnimi linijkami? Traktowane są one często jako część kolejnego żądania HTTP – zauważmy jednak, że nie jest ono zakończone (nie mamy na końcu dwóch ciągów CRLF). Atakujący wysyła więc kolejne żądanie, np. takie jak poniżej:

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

Żądanie ponownie jest dozwolone na serwerze frontend (URL to `/home`) i doklejane do „czekającego" żądania HTTP, które za moment będzie wysłane do backendu:

Listing 23. Zatrute żądanie HTTP. Odpowiedź na to żądanie jest wysyłana do atakującego

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

W listingu 23 widać cel wcześniejszego użycia enigmatycznego nagłówka: `Foo: x`. Bez niego po zatruciu żądania mielibyśmy **nagłówek** `GET /home HTTP/1.1`, co prawdopodobnie spowodowałoby odrzucenie całego żądania przez serwer HTTP z komunikatem błędu *Bad Request*.

POST: multipart/form-data

Czasem parametry wysyłane metodą `POST` korzystać będą właśnie z kodowania `multipart/form-data`. Przykład tego typu kodowania najczęściej znajdziemy w formularzach uploadujących pliki. Wygląda ono np. tak:

Listing 24. Żądanie z kodowaniem `multipart/form-data`

```
POST /request.php HTTP/2.1
Host: training.securitum.com
Content-Type: multipart/form-data; boundary=awnwWejh23kl
Content-Length: 323

--awnwWejh23kl
content-disposition: form-data; name="parametr1"

wartosc
jeden
--awnwWejh23kl
content-disposition: form-data; name="parametr2"

wartosc
dwa
--awnwWejh23kl
content-disposition: form-data; name="nasz_plik"; filename="file1.txt"
Content-Type: text/plain

Zawartosc pliku tekstowego
--awnwWejh23kl--
```

Warto zwrócić tutaj uwagę na tzw. ogranicznik (ang. *boundary*[28]). Najczęściej jest to losowa zawartość, choć, *nomen omen*, mająca pewne ograniczenia (np. posiada limit długości). Pełny ogranicznik to znaki CRLF (znak końca linii), dwa znaki minus (`--`) oraz wartość wskazana parametrem `boundary` w nagłówku. Czyli w naszym przykładzie jest to:

`--awnwWejh23kl` (oraz wcześniejszy znak końca linii).

Ostatni ogranicznik dodatkowo posiada jeszcze na końcu dwa kolejne znaki `--`. W naszym przykładzie to:

`--awnwWejh23kl--` (oraz wcześniejszy znak końca linii).

W skrócie, ogranicznik rozdziela ciało żądania na wiele (ang. *multipart*) części. W rozważanym przypadku mamy ich trzy. Każda z nich posiada nagłówek definiujący nazwę zmiennej: `content-disposition: form-data; name="nazwa"`. Następnie po dwóch znakach końca linii (CRLF) następuje przekazanie wartości zmiennej. Koniec wartości sygnalizowany jest przez ogranicznik. W listingu 25 widzimy zmienną o nazwie `parametr1` oraz jej wartość:

Listing 25. Parametr o nazwie `parametr1` w otoczeniu ograniczników (kodowanie `multipart/form-data`)

```
wartosc
jeden

--awnwWejh23kl
content-disposition: form-data; name="parametr1"

wartosc
jeden
--awnwWejh23kl
```

Pamiętajmy, że to tylko podstawy dotyczące tego typu kodowania parametrów. Jako przykład wykorzystania tego tematu do obchodzenia mechanizmów klasy WAF (*Web Application Firewall*) polecam pracę: *WAF Bypass Techniques – Using HTTP Standard and Web Servers' Behaviour*[29], w której można znaleźć sporo interesujących obejść bazujących właśnie na sprytnych modyfikacjach komunikacji HTTP.

Na koniec warto uzmysłowić sobie, że umieszczone w ciele żądania parametry mogą być przekazywane z wykorzystaniem metody `GET`. Pomysł wydaje się trochę dziwny, ale tego typu żądanie jest poprawne i co więcej, może czasem zostać użyte do wskazania poważnych podatności[30]. Przykład takiego żądania został przedstawiony w listingu 26.

Listing 26. Parametry w ciele żądania. Żądanie HTTP typu `GET`

```
GET /drupal-8.6.9/node/1?_format=hal_json HTTP/1.1
Host: 192.168.1.25
Content-Type: application/hal+json
Content-Length: 642

{
  "link": [
    {
      "value": "link",
      "options": "<SERIALIZED_CONTENT>"
    }
  ],
  "_links": {
    "type": {
      "href": "http://192.168.1.25/drupal-8.6.9/rest/type/shortcut/default"
    }
  }
```

Ciasteczka

Czyli popularne `cookie`, nazywane niekiedy niepoprawnie „plikami cookie”[31]. W kontekście naszych rozważań ciastka (mogące zawierać parametry) możemy znaleźć w nagłówku `Cookie` żądania HTTP. W jednym nagłówku może znaleźć się kilka różnych ciasteczek (mimo to nagłówek cały czas będzie nazywał się `Cookie`, nie `Cookies`). W którym miejscu ciastka są wysyłane z serwera? W odpowiedzi, w nagłówku `Set-Cookie`.

Zobaczmy przykład obu nagłówków. Po wpisaniu w przeglądarkę adresu: *http://training.securitum.com/cookie.php* serwer ustawia dwa ciastka (nagłówek odpowiedzi `Set-Cookie`):

Listing 27. Użycie nagłówka `Set-Cookie`

```
GET /cookie.php HTTP/1.1
Host: training.securitum.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:65.0)
Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
Date: Sun, 03 Feb 2019 16:58:11 GMT
Server: Apache
Set-Cookie: testowe-ciastko1=testowa-wartosc2
Set-Cookie: testowe-ciastko2=testowa-wartosc2
Vary: Accept-Encoding
Content-Length: 0
Connection: close
Content-Type: text/html
```

Kolejne zapytanie z przeglądarki jest już automatycznie uzupełnione o nagłówek `Cookie`:

Listing 28. Zapytanie automatycznie uzupełnione o nagłówek `Cookie`

```
GET /cookie.php HTTP/1.1
Host: training.securitum.com
Accept-Language: en-US,en;q=0.5
DNT: 1
Connection: close
Cookie: testowe-ciastko1=testowa-wartosc2; testowe-ciastko2=testowa-wartosc2
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

PODSUMOWANIE

Podstawowa wiedza dotycząca komunikacji HTTP na pewno przyda się w rzeczywistych eksperymentach z bezpieczeństwem aplikacji webowych. Warte zapamiętania jest to, że nie wszyscy kurczowo trzymają się specyfikacji HTTP, a czasem nawet błahe odstępstwa od utartych reguł mogą oznaczać brzemienne w skutkach wyniki[32]. Jest to prawdziwe wyzwanie dla osób zajmujących się bezpieczeństwem aplikacji WWW, a pogłębianie wiedzy na temat przypadków szczególnych i odstępstw od normy jest podstawową metodą doskonalenia warsztatu pentestera.

PRZYPISY:

1 *Hypertext Transfer Protocol Version 2 (HTTP/2)*, https://tools.ietf.org/html/rfc7540
2 *Hypertext Transfer Protocol Version 3 (HTTP/3)*, https://quicwg.org/base-drafts/draft-ietf-quic-http.html
3 *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, https://tools.ietf.org/html/rfc7230. W kontekście protokołu HTTP warto też spojrzeć na zbiorcze zestawienia specyfikacji: *HTTP Documentation*, https://httpwg.org/specs/ i: *HTTP resources and specifications*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications
4 *Simple Service Discovery Protocol* [w:] *Wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol
5 *CRLF* [w:] *Wikipedia, the free encyclopedia*, https://pl.wikipedia.org/wiki/CRLF. Por. też: wpis w komentarzach: *Kto zaproponuje jakieś zgrabne tłumaczenie CRLF?* [29.01.2019], https://www.facebook.com/sekurak/posts/2954887951204013
6 *Hypertext Transfer Protocol -- HTTP/1.1…*: rozdz. *5.1.1 Method*, https://tools.ietf.org/html/rfc2616#section-5.1.1
7 *Hypertext Transfer Protocol (HTTP) Method Registry*, https://www.iana.org/assignments/http-methods/http-methods.xhtml
8 *Uniform Resource Locators (URL)*, https://tools.ietf.org/html/rfc1738
9 Dotyczących również protokołu HTTP, zob. np.: *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, https://tools.ietf.org/html/rfc7230
10 *Uniform Resource Identifier (URI): Generic Syntax*, https://tools.ietf.org/html/rfc3986
11 *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*, https://tools.ietf.org/html/rfc3305
12 Przykład można znaleźć w: Kettle J., *Cracking the lens: targeting HTTP's hidden attack-surface*, rozdz. *Host overriding*, https://portswigger.net/blog/cracking-the-lens-targeting-https-hidden-attack-surface
13 Przykłady tego typu żądań można zobaczyć tutaj: inf0seq, *Directory Traversal in Axway File Transfer Direct*, https://inf0seq.github.io/cve/2019/01/20/Directory-Traversal-in-Axway-File-Transfer-Direct.html, oraz: Regel J., *[CVE-2017-7240] Miele Professional PG 8528 – Web Server Directory Traversal*, https://seclists.org/fulldisclosure/2017/Mar/63
14 Krawaczyński P., *Nagłówek X-Forwarded-For – problemy bezpieczeństwa…*, https://sekurak.pl/naglowek-x-forwarded-for-problemy-bezpieczenstwa/
15 Bentkowski M., *Jak w prosty sposób zwiększyć bezpieczeństwo aplikacji webowej*, https://sekurak.pl/jak-w-prosty-sposob-zwiekszyc-bezpieczenstwo-aplikacji-webowej/
16 Wnękowicz M., *Czy SSL szyfruje URL-e?*, https://sekurak.pl/czy-ssl-szyfruje-url-e/
17 Patrz też: *Referer header: privacy and security concerns*, https://developer.mozilla.org/en-US/docs/Web/Security/Referer_header:_privacy_and_security_concerns
18 Patrz: Abma J. (jobert), *Unauthenticated blind SSRF in OAuth Jira authorization controller*, https://hackerone.com/reports/398799
19 Patrz np.: O'Brien M. (mobsense), *Null dereference for invalid Host and If-Modified-* headers*, https://github.com/embedthis/appweb/issues/605
20 Domena: labs.data.gov, patrz: harisec, *SQL injection in https://labs.data.gov/dashboard/datagov/csv_to_json via User-agent*, https://hackerone.com/reports/297478
21 Więcej informacji można uzyskać np. tutaj: mpgn, *CVE-2019-5418 – File Content Disclosure on Rails*, https://github.com/mpgn/CVE-2019-5418, oraz Patterson A., *[CVE-2019-5418] File Content Disclosure in Action View*, https://groups.google.com/forum/#!topic/rubyonrails-security/pFRKI96Sm8Q
22 Patrz: *Uniform Resource Identifier (URI): Generic Syntax*, https://tools.ietf.org/html/rfc3986
23 Patrz: Montpas M.-A., *Content Injection Vulnerability in WordPress*, https://blog.sucuri.net/2017/02/content-injection-vulnerability-wordpress-rest-api.html
24 Patrz: Dalili S. *WAF Bypass Techniques – Using HTTP Standard and Web Servers' Behaviour*, https://www.slideshare.net/SoroushDalili/waf-bypass-techniques-using-http-standard-and-web-servers-behaviour
25 *Atvise WebMI2ADS Negative Content Length Vulnerability*,