Gynvael Coldwind

Diabeł tkwi w szczegółach: hosting plików

Projektując serwis internetowy, zdarza się, że chcemy zaoferować użytkownikom możliwość przesłania i udostępnienia plików, w szczególności zdjęć czy innych form graficznych (mogą to być np. fotografie przedmiotów w serwisach aukcyjnych, awatary na forach, zdjęcia profilowe itp.). Wydawać by się mogło, że jest to bardzo proste do zrobienia – wystarczy odebrać plik, zapisać na dysku serwera i udostępniać przez HTTP. Niestety, poprawne zaprojektowanie tego typu systemu od strony bezpieczeństwa serwisu i użytkowników jest trudniejsze niż mogłoby się wydawać – a diabeł, jak zwykle, tkwi w szczegółach.

W niniejszym artykule przez termin *hosting plików* rozumiem przechowywanie plików otrzymanych od użytkowników oraz ich udostępnianie w serwisie internetowym przez protokół HTTP lub HTTPS. W szczególności skupię się na plikach graficznych takich jak JPEG, PNG czy GIF, ponieważ na hostingu takich właśnie plików najczęściej zależy twórcom serwisów internetowych, a jednocześnie pozwalają one na omówienie dodatkowych, bardziej specyficznych, problemów. Niemniej jednak większość rozważań można uogólnić na dowolny rodzaj plików.

Przed przejściem do meritum dodam jeszcze, że artykuł ma na celu wskazanie różnych, nie zawsze poprawnych, spotykanych i stosowanych metod, wraz ze wskazaniem ich słabych stron. Warto więc doczytać tekst do końca przed przystąpieniem do jakichkolowiek działań. Zachęcam również do sięgnięcia do dodatkowych źródeł w razie gdyby któryś z użytych terminów był nieznany.

ZAGROŻENIA

Zacznijmy jednak od początku, czyli od pytania, czy warto w ogóle poświęcać czas na problem bezpiecznego hostingu plików? Lub, zadając pytanie w trochę inny sposób, co ryzykujemy, jeśli zaniedbamy ten problem?

Problemy wynikłe z zaniedbania możemy podzielić na dwie kategorie: problemy po stronie serwera (ang. *server-side*), oraz po stronie klienta, czyli w tym przypadku przeglądarki internetowej (ang. *client-side*).

SERVER-SIDE

Pierwszym problemem jest rozpoznawanie po rozszerzeniu plików z kodem do wykonania - sztandarowym przykładem jest tu PHP w najczęściej spotykanej konfiguracji. Wystarczy, aby plik miał rozszerzenie *.php* (np. zdjecie_z_wakacji.php) oraz możliwe było bezpośrednie odwołanie się do niego przez HTTP (np. wchodząc na stronę http://example.com/uploads/zdjecie_z_wakacji.php) - w takim wypadku serwer HTTP wywoła interpreter PHP, który z kolei wykona cały kod zawarty w danym pliku. Z jednej strony jest to wygodne, ale z drugiej stwarza problemy, jeśli użytkownik o niecnych zamiarach (dalej: atakujący) zdołałby doprowadzić do zapisania na serwerze pliku o rozszerzeniu *.php* oraz spowodować jego wykonanie przez interpreter.

Drugi problem dotyczy błędów klasy **LFI** (patrz ramka), których eksploitacja staje się trywialna, jeśli atakujący ma możliwość umieścić wybrany plik na atakowanym serwerze. Przykładowo: atakujący może stworzyć plik z kodem PHP, nadać mu rozszerzenie *.jpg*, przesłać na serwer, korzystając z możliwości upload'u zdjęć (plików), po czym wykorzystać przesłany plik przy LFI, powodując wykonanie własnego kodu z uprawnieniami interpretera.

LFI (ang. *Local File Inclusion*)

Najłatwiej wytłumaczyć LFI na przykładzie:

```
<?php include "pages/" . $_GET['page']; ?>
```

Powyższy kod *server-side* strony WWW ma w założeniu służyć do wyświetlenia podstrony znajdującej się w katalogu *pages*. Przykładowo, jeśli użytkownik wejdzie na stronę http://example.com/showpage.php?page=contact.html, nastąpi załadowanie oraz wykonanie (dyrektywa *include* w PHP powoduje wykonanie podanego w parametrze pliku przez interpreter PHP; stąd też słowo "*Inclusion*" w nazwie tej klasy błędów) pliku *pages/contact.html*, co spowoduje zapewne wygenerowanie i przesłanie przeglądarce odpowiedniego dokumentu HTML.

Oczywiście parametr *page* w powyższym przykładzie nie jest w żaden sposób walidowany, a więc nic nie stoi na przeszkodzie, aby atakujący przypisał mu nieprzewidzianą przez programistę wartość, np. *?page=../uploads/evil.jpg*, powodując tym samym wykonanie kodu PHP znajdującego się we wskazanym pliku lub wyświetlenie jego zawartości.

Kolejny problem dotyczy **DoS** (ang. *Denial of Service*, pl. odmowa usługi) – atakujący, poprzez przesłanie jednego lub większej ilości bardzo dużych plików, może spowodować wyczerpanie dostępnego miejsca na dyskach serwera, co z kolei może prowadzić do utraty danych, zaprzestania tworzenia logów czy błędnego działania niektórych elementów serwisu internetowego (w szczególności mechanizmu przesyłania plików).

Ostatni problem dotyczy przetwarzania obrazów po stronie serwera (np. zmiany ich wielkości czy innych modyfikacji wymagających pełnego wczytania danych obrazu z pliku). W przypadku niskiej jakości kodu odpowiedzialnego za parsing formatów graficznych napisanego w języku, w którym łatwo o błąd (jak np. C czy C++ - patrz dwa poprzednie artykuły z serii "Diabeł tkwi w szczegółach"), atakujący może pokusić się o próbę znalezienia luki i jej eksploitacji. Z kolei udana eksploitacja może doprowadzić do zdobycia przez atakującego uprawnień procesu przetwarzającego zdjęcia. Tematyka poprawnego przetwarzania danych wykracza jednak poza zakres niniejszego artykułu; jako niezbędne minimum zaznaczę, że proces przetwarzający pliki otrzymane od użytkowników powinien być odpowiednio odizolowany od reszty środowiska [1].

CLIENT-SIDE

Po stronie klienta (czyli przeglądarki internetowej) problemem jest w zasadzie tylko potencjalna możliwość wykonania kodu JavaScript należącego do atakującego w kontekście naszego serwisu - czyli tzw. XSS (patrz ramka). Jeśli atakujący zdoła umieścić dowolny plik HTML (np. .html), SVG (np. .svg),

XML (np. .xml), Adobe Flash (np. .swf) czy aplet JAVA (np. .jar) w domenie naszego serwisu, to, poprzez wykonanie kodu znajdującego się w tych plikach w przeglądarkach innych użytkowników naszego serwisu, może uzyskać dostęp do ich ciasteczek (w tym ciasteczek sesji).

XSS (ang. Cross Site Scripting)

Domyślnie strony internetowe są od siebie odseparowane zgodnie z tzw. *Same Origin Policy* [2] - jest to zbiór zasad mówiących, w uproszczeniu, że kod JavaScript uruchomiony w kontekście jednego serwisu (domeny) nie ma dostępu do zawartości i ciasteczek stron należących do innych serwisów (domen).

Błędem XSS określana jest sytuacja, w której atakujący może doprowadzić do uruchomienia dostarczonego przez siebie kodu JavaScript w kontekście atakowanego serwisu. Udana eksploitacja błędów typu XSS daje atakującemu dostęp do zawartości strony, ciasteczek (w tym ciasteczek sesji używanch m.in. do kontroli dostępu), oraz umożliwia wykonywanie większości akcji w serwisie z uprawnieniami zalogowanego użytkownika.

Mając zdefiniowane poszczególne problemy, możemy rozważyć kilka różnych rozwiązań w kolejnych momentach przesyłania pliku oraz jego udostępniania – zaczynając od formularza używanego do upload'u, poprzez sam proces przesyłania danych, aż do przetwarzania odebranych danych i ich dalszego hostingu.

CLIENT-SIDE: FORMULARZ UPLOADU

Tworząc w HTMLu formularz, można pokusić się o zdefiniowanie, jakiego rodzaju pliki mogą być przesyłane (parametr *accept* dla tagu <*input*> [3]) – tak, aby mieć pewność, że użytkownik będzie mógł przesłać jedynie pliki graficzne.

Dodatkowo niektóre technologie, jak np. PHP, umożliwiają w tym miejscu zdefiniowanie maksymalnej wielkości przesyłanego pliku - w PHP odpowiedzialny za to jest dodatkowy, ukryty <*input*> o nazwie *MAX_FILE_SIZE* [4].

Przykładowy fragment formularza stworzonego według powyższych zasad wygląda następująco:

```
<input type="hidden" name="MAX_FILE_SIZE" value="2097152" />
<input type="file" name="f" accept=".jpg,.jpeg,.png,.
gif,image/*" />
```

Niestety, nie tędy droga - powyższy kod nie ma żadnego znaczenia z punktu widzenia bezpieczeństwa - atakujący może go dowolnie zmodyfikować w swojej przeglądarce, usuwając wszelkie ograniczenia (np. za pomocą Firebug, lub podobnych narzędzi), czy też stworzyć własną wersję formularza wskazującego na nasz skrypt/serwlet od upload'u.

Projektując serwis WWW, należy pamiętać, aby nie umieszczać zabezpieczeń tylko po stronie klienta, ponieważ kod każdego takiego zabezpieczenia atakujący może zmodyfikować lokalnie wedle własnego uznania (jest to typowe dla architektur klient-serwer). Zabezpieczenia client-side mają raczej charakter kosmetyczny i mają na celu na przykład poinformowanie użytkownika, że dany rodzaj pliku nie jest obsługiwany, zanim dojdzie do faktycznej próby upload'u.

Skoro nic nie zdziałamy po stronie przeglądarki, przejdźmy do server-side.

SERVER-SIDE: ODBIERANIE PRZESYŁANEGO PLIKU

Przeglądarka internetowa, oprócz zawartości danego pliku, przesyła przed danymi dwie dodatkowe informacje: nazwę pliku (np. *lolcat.gif*) oraz jego typ MIME (parafrazując Wikipedię, jest to dwuczęściowy identyfikator typu danych stosowany m.in. w protokole HTTP [13]; np.: *image/gif*). Obie informacje można potencjalnie wykorzystać do weryfikacji tego, czy przesyłany plik jest rzeczywiście plikiem graficznym - w takim wypadku typ MIME rozpoczyna się od "*image/*" – a także sprawdzić, czy jego rozszerzenie nie stanowi zagrożenia, czyli jest na przykład równe ".jpg" czy ".png"[1]. Niestety, sprawdzenie typu MIME nic nie da, ponieważ informacja ta pochodzi od klienta, a atakujący może przesłać serwerowi, co tylko mu się podoba (np. *image/gif*, nawet jeśli plik z formatem GIF nie ma nic wspólnego). Również nazwę pliku powinniśmy traktować jak niezaufany string, który może nie być poprawnie sformatowaną nazwą pliku.

Co może się stać, gdybyśmy jednak zdecydowali się zaufać otrzymanej nazwie pliku i wykorzystać ją jako część/całość nazwy pliku docelowego zapisywanego na dysk serwera?

Przede wszystkim, atakujący może próbować zmienić miejsce, w którym zostanie zapisany plik docelowy, dodając np. "../../../inny/katalog/" na początku przesłanej nazwy pliku (jest to tzw. *path traversal attack*). Po drugie, może starać posłużyć się tzw. *Poison Null Byte* (patrz ramka) lub podobnym sposobem do oszukania skryptu weryfikującego rozszerzenie (należy natomiast zauważyć, że wykorzystanie *Poison Null Byte* w tym miejscu nie zawsze jest możliwe i zależy od szczegółów implementacyjnych używanych technologii). Tak więc w skrajnie złym wypadku doprowadziłoby to do np. stworzenia pliku *.php* w katalogu dostępnym via HTTP, a to z kolei spowodowałoby wykonanie przez serwer kodu atakującego.

Należy również pamiętać, że nazwa pliku może zawierać znaki typu < czy > – a więc, jeśli zdecydujemy się wyświetlić nazwę przesłanego pliku na podstronie naszego serwisu, należy zadbać o odpowiednie kodowanie takich znaków, tak aby nie doprowadzić do powstania błędu typu XSS.

Poison Null Byte

Poison Null Byte (pl. trujący bajt zerowy) jest techniką polegającą na celowym wstawieniu bajtu zerowego w string typu Pascal (string reprezentowany wewnętrznie jako para: długość danych oraz dane), w celu przedwczesnego zakończenia danego ciągu znaków podczas jego konwersji na string w stylu C (czyli terminowany bajtem zerowym, oznaczanym zazwyczaj '\0').

Typowym przykładem jest użycie *Poison Null Byte* do "odcięcia" rozszerzenia w nazwie pliku. Rozważmy poniższy kod:

```
<?php readfile("./" . $_GET['filename'] . ".txt"); ?>
```

W zamierzeniu programisty kod ten ma umożliwić wyświetlenie zawartości jedynie plików o rozszerzeniu ".txt". Atakujący ma jednak możliwość wstawić bajt zerowy w parametr filename (korzystając w tym wypadku z *URL encoding*), np.:

▶ http://example.com/?filename=another_file.php%oo

Spowoduje to powstanie wewnętrznie w maszynie wirtualnej PHP ciągu "./another_file.php\o.txt", który, bezpośrednio przed wywołaniem niskopoziomowego API otwierającego plik jak np. fopen czy CreateFile, zostanie skonwertowany na C-string "./another_file.php".

Niektóre języki (jak np. najnowsza wersja PHP czy Python w wersji 2.X) generują wyjątek, jeśli wykryją *Poison Null Byte* w konwertowanym stringu. W innych (np. Java) programista musi zbadać występowanie bajtu zerowego przed podaniem danego ciągu otrzymanego od użytkownika do funkcji, które wewnętrznie dokonają konwersji ciągu na C-string.

Podsumowując, nie należy ufać ani przesłanej nazwie pliku, ani przesłanemu typowi MIME. W szczególności nie należy wykorzystywać otrzymanej nazwy pliku do zapisu pliku na dysku (chyba że dokonamy odpowiedniej walidacji i usuniemy wszystkie znaki poza dopuszczalnymi). Zazwyczaj dobrze jest wygenerować losową, unikalną nazwę dla pliku.

Należy jednak wskazać, że odbieranie danych to dobry moment na kontrolę wielkości pliku, ponieważ w tym momencie (podczas przesyłania) możemy odpowiedzieć na pytanie, czy przesyłany plik jeszcze mieści się w dopuszczalnym limicie wielkości, czy może już go przekroczył i należy wyświetlić stronę błędu.

1 Oczywiście przeciwne podejście (typu *blacklist*), czyli sprawdzanie, czy rozszerzenie jest różne od ".php" czy ".html", jest nieefektywne, ponieważ wystarczy, aby atakujący znalazł jedno rozszerzenie, o którym nie pomyśleliśmy, jak np. ".php5" czy ".htm" dla wspomnianego przykładu, aby zabezpieczenie stało się bezwartościowe.

SERVER-SIDE: SPRAWDZENIE POPRAWNOŚCI PLIKU

W tym momencie odebraliśmy już cały plik (wykluczając przy tym atak typu DoS) oraz wygenerowaliśmy dla niego losową nazwę, tym samym zapobiegając problemowi związanemu z rozszerzeniami plików.

Czy zatem wyeliminowaliśmy również kwestie podatności XSS (w końcu wiemy już, że plik nie będzie miał rozszerzenia *.html*)? Niestety nie – jak się okazuje, rozszerzenie pliku nie jest prawie w ogóle istotne dla przeglądarki. Najbardziej istotny jest typ MIME, z którym plik jest serwowany – a ten możemy ustawić odpowiednio według ustalonego, faktycznego typu danych. Niemniej jednak, niektóre starsze przeglądarki (w szczególności IE6, która ma nadal wierną, choć malejącą, grupę użytkowników) w niektórych szczególnych wypadkach ignorują typ MIME otrzymany od serwera, np. jeśli w początkowej części pliku znalazł jakikolwiek poprawny tag HTML. Warto dodać, że mechanizm próbujący rozpoznać typ zasobu po jego zawartości zwany jest w świecie przeglądarek *content sniffing*, i jest w różnorodny sposób zaimplementowany w niektórych przeglądarkach [14].

Nie rozwiązaliśmy jeszcze również problemu łatwej eksploitacji błędów typu LFI – w końcu przesłany plik może zawierać kod PHP mimo rozszerzenia *.jpg* czy *.gif*.

Potencjalnym rozwiązaniem jest więc upewnić się, że przesłany plik jest faktycznie plikiem graficznym - w końcu wydawać by się mogło, że dokument HTML czy kod PHP nie mogą być jednocześnie poprawnymi plikami graficznymi. Podążając tym tropem, optymalnie byłoby wykorzystać do tego gotową funkcję. Przykładowo, w języku PHP często polecaną do tego celu funkcją jest *getimagesize* [5], która na wejściu przyjmuje nazwę pliku, a zwraca tablicę zawierającą m.in. wywnioskowany typ MIME oraz wielkość bitmapy, lub wartość *FALSE* w przypadku niepowodzenia. Wydawałoby się więc, że funkcja ta jest idealna do naszych celów.

Niestety, analizując kod źródłowy funkcji *getimagesize*, łatwo zauważyć, że cała "weryfikacja" pliku graficznego opiera się na porównaniu kilku początkowych bajtów ze znaną sygnaturą [6] [7] [8]. A więc, aby oszukać funkcję i móc umieścić dalej np. HTML czy kod PHP (czyli tzw. *payload*), wystarczy, aby atakujący skopiował kilka pierwszych bajtów poprawnej sygnatury na początek preparowanego pliku.

Można więc do sprawdzenia poprawności pliku użyć funkcji, która w pełni ładuje (parsuje) plik graficzny. Okazuje się jednak, że jest to również niewystarczające, z kilku powodów.

Po pierwsze, atakujący może przesłać bardzo mały poprawny plik graficzny z doklejonym na końcu payloadem. Ewentualnie, dodatkowe dane atakujący może umieścić w samym pliku, np. w sekcji komentarza (zarówno PNG, GIF, jak i JPEG to umożliwiają), w dowolnej nieużywanej przestrzeni w pliku (w sporej ilości formatów graficznych da się taką znaleźć lub stworzyć), lub wręcz w samych danych bitmapy. Idealnym przykładem w tym miejscu jest GIFAR (patrz ramka).

GIFAR

GIFAR jest tzw. binarnym plikiem poliglotycznym (czyli plikiem zgodnym ze specyfikacją więcej niż jednego formatu binarnego), powstałym ze złożenia pliku graficznego GIF oraz archiwum JAR (de facto archiwum ZIP). Jest to możliwe, ponieważ nagłówek formatu GIF znajduje się na samym początku pliku, natomiast nagłówek (a w zasadzie "stopka") pliku ZIP jest umieszczony pod koniec pliku, a więc nagłówki (dotyczy to również danych) nie rywalizują ze sobą o te same pozycje w pliku. GIFAR jest więc jednocześnie w pełni poprawnym plikiem GIF, jak i w pełni poprawnym plikiem ZIP.

Z uwagi na powyższą charakterystykę, w przypadku nieprawidłowo zabezpieczonego serwisu internetowego możliwy jest upload pliku GIF zawierającego aplet Java, który zostanie zwalidowany jako poprawny plik graficzny, a następnie uruchomienie w przeglądarce danego apletu w kontekście danego serwisu (czyli do XSS).

Po raz pierwszy metoda XSS korzystająca z GIFAR została opisana przez Billi'ego Riosa, Roba Cartera, Johna Heasmana i Nate McFetersa w roku 2008 [15].

Z podobnych binarnych plików poliglotycznych warto zobaczyć JPEG+HTML [16] oraz CorkaMIX czyli EXE+PDF+JAR+HTML [17].

Można więc próbować iść o krok dalej i np. zmienić wielkość bitmapy, po czym zapisać ją do nowego pliku na dysku - w ten sposób usunięte zostaną wszystkie dane nie będące bezpośrednio związane z obrazem, a przez zmianę wielkości czy ponowny zapis (*encoding*) danego formatu przy użyciu innego enkodera z innymi ustawieniami, zniszczymy wszelki ewentualny payload obecny w oryginalnym pliku.

Jednak, jak pokazuje praktyka, jeśli atakujący ma wystarczająco wiele informacji o systemie, który atakuje, lub jest w stanie wywnioskować potrzebne informacje przesyłając kolejne pliki i obserwując zmiany, może spróbować stworzyć taki plik wejściowy, by po dokonanych zmianach oraz zapisie na dysk znalazł się w nim oczekiwany payload (nie zawsze jest to jednak trywialne; zainteresowanych czytelników zachęcam do zajrzenia np. do [9]).

Oczywiście można próbować iść jeszcze dalej i wprowadzać losowe parametry dla enkodera lub wręcz dodawać szum do danych obrazu i palety kolorów (jeśli takowa jest w użyciu), jednak spowoduje to utratę jakości i zwiększenie zużycia miejsca na dysku (losowy szum bardzo słabo się kompresuje), a więc to rozwiązanie nie jest najbardziej praktyczne. Również, nie do każdego rodzaju danych można wprowadzić szum czy losowość – przykładem mogą być tu pliki podpisane cyfrowo lub zaszyfrowane.

Można też każdy plik spróbować "zaszyfrować" losowym kluczem, jednak spowoduje to zwiększenie kosztu wydajnościowego przy odczycie po stronie serwera oraz nie rozwiązuje problemu XSS, ponieważ dane i tak muszą zostać odszyfrowane przed dostarczeniem do klienta.

Jak więc poradzić sobie z LFI? Tak naprawdę nie tędy droga. Idealnie byłoby po prostu nie mieć w kodzie błędów typu LFI.

Drugą, i zalecaną w tym miejscu, linią obrony jest taka konfiguracja danej technologii (np. środowiska PHP w taki sposób), aby dany skrypt/serwlet/itp. nie mogły czytać plików spoza katalogu projektu. W takim wypadku przesyłane przez użytkowników pliki powinny być umieszczane poza katalogiem projektu (w idealnym wypadku na innym serwerze).

Czytelnikom zainteresowanym innymi możliwościami eksploitacji LFI chciałbym wskazać pierwszą część artykułu "*PHP LFI to arbitratry code execution via rfc1867 file upload temporary files*" [10].

SERVER-SIDE: SERWOWANIE ZASOBU GRAFICZNEGO

Ostatnim elementem systemu jest serwowanie danego zasobu graficznego przez HTTP w naszym serwisie, a więc zazwyczaj (błędnie – o tym za chwilę) z domeny naszego serwisu. Natomiast, z zagrożeń, z którymi musimy się rozprawić, został jeszcze tylko XSS.

Zacząć należy od poprawnego ustawienia typu MIME (tak aby przeglądarki nie próbowały na własną rękę wnioskować o typie danych), zgodnego z faktycznym typem pliku, np.:

```
Content-Type: image/gif
```

Niestety, to nie wystarczy - jak wspomniałem wcześniej, niektóre starsze przeglądarki mimo wszystko skorzystają z mechanizmu *content sniffing*. Można spróbować przekonać przeglądarkę, aby dla danego zasobu nie korzystała w ogóle z tego mechanizmu - służy do tego nagłówek *X-Content-Type-Options* [11]:

```
X-Content-Type-Options: nosniff
```

Niestety nie wszystkie przeglądarki i nie wszystkich ich wersje mają zaimplementowaną obsługę niestandardowego powyższego nagłówka. Można więc próbować wymusić ściągnięcie pliku w przypadku jego bezpośredniego otwarcia (zamiast go pokazywać) – służyć do tego może nagłówek *Content-Disposition* [12], np.:

```
Content-Disposition: attachment; filename="..."
```

Również w tym wypadku okazuje się, że niektóre przeglądarki nie implementują poprawnie powyższego nagłówka i w pewnych wyjątkowych sytuacjach starają się mimo wszystko wyświetlić zasób zamiast po prostu go pobrać.

Podsumowując powyższe rozwiązania, nie wszystkie istniejące na komputerach użytkowników przeglądarki będą respektować w pełni powyższe nagłówki. Dodatkowym problemem są popularne pluginy do przeglądarek, jak Flash czy Java, których niektóre wersje również zignorują nieprawidłowy *Content-Type* dla apletu/obiektu. A więc, mimo zastosowania wszystkich opisanych w niniejszym artykule technik nie jesteśmy w stanie w pełni bezpiecznie serwować plików otrzymanych od użytkowników.

Jak więc rozwiązać problem XSS podczas serwowania plików przesłanych przez użytkownika z domeny naszego serwisu? Odpowiedź może zaskoczyć czytelnika: na obecny stan wiedzy nie jest to możliwe w praktyce, właśnie z uwagi na różnorodne zachowania obecnych na rynku przeglądarek.

Możliwe jest natomiast obejście problemu. Zauważmy, że XSS jest dla nas problemem, ponieważ umożliwia kradzież sesji, a konkretniej ciasteczek sesji, związanych z domeną naszego serwisu. Serwowanie zasobów przesłanych przez użytkowników z innej domeny jest więc intuicyjnym rozwiązaniem. Istotne jest, aby domena ta nie ustawiała żadnych ciasteczek - w końcu nie można ukraść ciasteczek sesji, jeśli ich nie ma – oraz nie była subdomeną głównej domeny serwisu.

Warto zauważyć, że z tej właśnie metody korzystają duże serwisy. Na przykład serwis allegro.pl posługuje się domeną allegroimgproducts.pl do serwowania zdjęć przedmiotów, serwis ebay.pl korzysta z domeny ebayimg.com, serwis wykop.pl korzysta z imgwykop.pl, serwis pl.wikipedia.org korzysta z upload.wikimedia.org itd.

Oczywiście stosując powyższy schemat, pewnym problemem staje się kontrola dostępu, jeśli dostęp do danego zasobu ma być chroniony. Tematyka ta wykracza jednak poza zakres niniejszego artykułu, więc wspomnę tylko krótko, że długie, losowe ścieżki do zasobów, wykluczenie całej domeny z indeksowania w wyszukiwarkach oraz ograniczone czasowo tokeny dostępu do zasobów są kierunkiem, który należy rozważyć.

PODSUMOWANIE

Przede wszystkim nie należy ufać niczemu, co przychodzi z zewnątrz, a w szczególności nazwie pliku czy informacji o jego typie MIME. Również wszelkie ograniczenia co do typu pliku czy jego wielkości powinny być zaimplementowane przede wszystkim po stronie serwera. Przesadna weryfikacja typu pliku czy jego ponowny enkoding nie ma praktycznego sensu (z punktu widzenia bezpieczeństwa) – zamiast tego, należy umieszczać otrzymane pliki w takim miejscu, by nie było możliwe wykonanie zawartego w nich payload'u przez interpreter / maszynę wirtualną technologii, z której korzystamy. Otrzymane pliki najlepiej jest serwować z innej, "bezciasteczkowej" domeny.

Mam nadzieję, że niniejszy tekst w pewnym stopniu przybliżył czytelnikowi tematykę bezpiecznego hostingu plików.

Wszelkie opinie wyrażone w artykule są prywatnymi opiniami autora

W sieci

- [1] http://en.wikipedia.org/wiki/Sandbox_(computer_security)
- [2] http://en.wikipedia.org/wiki/Same_origin_policy
- [3] http://dev.w3.org/html5/spec/states-of-the-type-attribute.html#attr-input-accept
- [4] http://www.php.net/manual/en/features.file-upload.post-method.php
- [5] http://www.php.net/manual/en/function.getimagesize.php
- [6] http://gynvael.coldwind.pl/?id=219
- [7] http://gynvael.coldwind.pl/?id=220
- [8] http://gynvael.coldwind.pl/?id=226
- [9] http://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/
- [10] http://gynvael.coldwind.pl/download.php?f=PHP_LFI_rfc1867_temporary_files.pdf
- [11] http://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx
- [12] http://www.w3.org/Protocols/rfc2616/rfc2616-sec19.html
- [13] http://pl.wikipedia.org/wiki/Typ_MIME
- [14] http://code.google.com/p/browsersec/wiki/Part2#Survey_of_content_sniffing_behaviors
- [15] http://xs-sniper.com/blog/2008/12/17/sun-fixes-gifars/
- [16] http://lcamtuf.coredump.cx/squirrel/
- [17] http://code.google.com/p/corkami/wiki/mix

Gynvael Coldwind gynvael@coldwind.pl

Na co dzień autor pracuje w firmie Google na stanowisku Information Security Engineer. Po godzinach prowadzi bloga oraz nagrywa podcasty o programowaniu (http://gynvael.coldwind.pl/). Hobbystycznie programuje od ponad 20 lat (w tym ponad 10 lat w C i C++).