Gynvael Coldwind Mateusz Jurczyk

Programistyczne potknięcia

Jak ich unikać?

PWN

Spis treści

- **Wstęp**.....3
- **Arytmetyka i obliczenia**.....4
  - Przekroczenie zakresu liczb całkowitych.....4
  - Obcięcie zakresu liczb całkowitych.....8
- **Wycieki danych.**.....12
  - Głębokie ukrycie.....12
  - Użycie niezainicjalizowanej pamięci.....13
- **Stabilność i niezawodność**.....20
  - Kolizje w tablicach haszujących.....20
  - Wyszukiwanie wszystkiego.....22
- **Wielowątkowość i wielozadaniowość**.....24
  - Wyścigi w systemie.....24
- **Dane wejściowe**.....27
  - Deserializacja niezaufanych danych.....27
  - Logika aplikacji.....31
  - Oscylatory walutowe.....31
  - Dziwactwa.....33
  - Nierówne równości.....33
- **Kryptografia**.....35
  - Wartości (nie)losowe.....35
- **Przenośność kodu**.....37
  - Nazwy plików.....37
- **Zakończenie**.....39
- **Bibliografia**.....40

Wstęp

Historia tworzenia i testowania oprogramowania jest ściśle związana z historią elektronicznych maszyn liczących, której początki sięgają lat '30 i '40 poprzedniego wieku, kiedy to sławne postaci komputerowego świata takie jak Alan Turing, John von Neumann czy Tommy Flowers tworzyli podwaliny współczesnej informatyki. Od czasu powstania pierwszej programowalnej maszyny *software* stał się nieodłączną częścią każdego komputera, którego rozwój niedługo później wyewoluował do niezwykle rozległej i skomplikowanej dziedziny nauki w swoich podstawach odrębnej od *hardware*, czyli fizycznie istniejącego sprzętu, na którym jest wykonywany. Ze względu na ludzką omylność oraz fakt, że programowanie jest pod wieloma względami procesem trudnym, wymagającym koncentracji, wiedzy oraz doświadczenia, błędy (tzw. *bugi*) wszelkich rodzajów towarzyszyły i towarzyszą po dziś dzień twórcom oprogramowania, stając się wręcz integralną i akceptowaną częścią ich pracy.

Błędy znajdujące się w programach komputerowych można dzielić na wiele różnych kategorii: błędy w składni lub logice aplikacji, błędy wpływające na stabilność lub bezpieczeństwo systemu, błędy ujawniające się w trakcie pisania kodu, po tygodniach, latach, a nawet dekadach! W historii ostatnich lat znajdziemy błędy które w żaden znaczący sposób nie wpłynęły na losy świata jak również takie, których skutki były tragiczne, a firmy i państwowe organizacje obciążyły stratami rzędu milionów, a nawet miliardów dolarów. Naturalnym wydaje się, że nie sposób całkowicie wyeliminować wszystkie usterki w programach sterującymi aparatami, komórkami, komputerami domowymi, reaktorami jądrowymi czy sztucznymi satelitami; możemy jednak minimalizować liczbę popełnianych pomyłek oraz naprawiać błędy, zanim znajdą się one w kodzie produkcyjnym. W niniejszym artykule przedstawiamy przykłady kilkunastu najciekawszych, zdaniem autorów, rodzin błędów i programistycznych potknięć, wraz z wyjaśnieniem ich natury, możliwości unikania oraz powiązanymi przykładami z życia. Zapraszamy do lektury!

Arytmetyka i obliczenia

Przekroczenie zakresu liczb całkowitych

Zaczniemy od bardzo prostego błędu, który spotyka się w wielu językach programowania - przekroczenia zakresu liczb całkowitych, lub częściej *integer overflow* (dosłownie *przepełnienie liczby całkowitej*). Przykłady kodu zawierającego błąd tego typu w różnych językach programowania przedstawione są poniżej:

```
// C, C++
unsigned short dlugosc_tekstu = OdbierzDlugoscTekstu(socket);
char *bufor = (char*)malloc(dlugosc_tekstu + 1);
OdbierzTekst(bufor, dlugosc_tekstu, socket);
bufor[dlugosc_tekstu] = '\0';

// ActionScript
var szerokosc:int = PobierzInt(plik);
var wysokosc:int = PobierzInt(plik);
var wielkosc:int = szerokosc * wysokosc;
```

W niektórych językach programowania takich jak C, C++, Objective-C, ActionScript czy Java, zmienne typu całkowitego mają ograniczony rozmiar (zazwyczaj wyrażany w bitach; np. typ *int* w ActionScript ma wielkość 32 bitów), a co za tym idzie, ograniczony zakres wartości które można w nich przechować. W zasadzie dowolna książka traktująca o danym języku programowania rozpoczyna opis dostępnych typów zmiennych od podania zakresu wartości, które można w nich przechować; w przypadku wspomnianego typu *int* w ActionScript będą to wartości od -2^{31} do $2^{31}-1$ (czyli od około minus dwóch miliardów, do około dwóch miliardów).

Powstaje więc pytanie - co się stanie, jeśli program w toku wykonywania przekroczy dozwolony zakres zmiennej, w rezultacie wykonanej właśnie operacji arytmetycznej? Możliwości jest oczywiście kilka i zależą one zarówno od konkretnej technologii, jej wersji, typu zmiennej, jak i systemu, czy architektury procesora na którym dany program jest uruchamiany. W zdecydowanej większości przypadków do czynienia mamy z tzw. arytmetyką "modulo" która, rozpatrując kwestię z niskopoziomowej perspektywy,

polega na "przycięciu" wyniku operacji do dolnych N-bitów, które mieszczą się w zakresie danego typu (dla typów liczb naturalnych jest to równoważne z wykonaniem dzielenia modulo przez maksymalną wartość, którą można pomieścić w zmiennej powiększoną o jeden).

Rozważmy ten problem na podanym powyżej przykładzie z ActionScript - jeśli funkcja `PobierzInt` zwróciłaby w obu przypadkach wartość 70 000, to wynikiem mnożenia byłoby oczywiście 4 900 000 000 (niecałe pięć miliardów). Binarnie (w systemie dwójkowym) można tę liczbę zapisać jako:

```
1 0010 0100 0001 0000 0001 0001 0000 0000
```

Niestety, liczba całkowita tego rzędu wymaga do wyrażenia 33 bitów, a zadeklarowana w przykładzie zmienna `wielkosc` ma rozmiar jedynie 32 bitów. Z tego powodu najbardziej znacząca część wyniku zostanie odrzucona:

```
1 0010 0100 0001 0000 0001 0001 0000 0000
```

W efekcie otrzymujemy znacznie mniejszą liczbę - 605 032 704 - która nijak ma się do oczekiwanego wyniku.

Błędy tego typu występują w oprogramowaniu niezwykle często, jednak jedynie w nielicznych przypadkach objawiają się podczas codziennej pracy - z reguły programy w normalnych warunkach nie mają "okazji" operować na danych wystarczająco obszernych, by doprowadzić do przepełnienia zmiennej typu wybranego przez programistę. Błąd tego typu może zostać jednak celowo wywołany przez osobę o niecnych zamiarach, która, będąc w stanie spreparować nieprzewidziane przez autora warunki działania programu, może doprowadzić do wykonania kontrolowanego przez nią kodu w kontekście podatnej aplikacji (czyli do przejęcia nad nią kontroli). W związku z tym przekroczenie zakresu zmiennej, szczególnie w przypadku języków C oraz C++ (choć nie tylko), może prowadzić do poważnych problemów związanych z bezpieczeństwem aplikacji, a w konsekwencji całego systemu komputerowego.

CIEKAWOSTKA

Za przykład może posłużyć tutaj *exploit* zaprezentowany przez hakera o pseudonimie *PinkiePie* na konferencji PacSec w Tokio w listopadzie 2013. PinkiePie, wykorzystując właśnie błąd przepełnienia zmiennej typu całkowitego w przeglądarce Google Chrome (a następnie kilka kolejnych błędów), był w stanie wykonać dowolny kod na smarfonach Nexus 4 oraz Samsung Galaxy S4, tym samym zdobywając nagrodę w wysokości 50,000 USD [1].

Problemy związane z przepełnieniami typu całkowitego można rozwiązać na kilka sposobów:

- Niektóre technologie umożliwiają wykrycie przepełnienia w momencie jego wystąpienia; na przykład w języku ADA w takiej sytuacji zostanie rzucony wyjątek (w przypadku kompilatora z rodziny GCC wymaga to kompilacji z opcją -gnato), który następnie może być obsłużony przez aplikację. Innym przykładem jest asembler architektury x86, w którym flagi CF (ang. *Carry Flag*, tzw. flaga przeniesienia dla liczb naturalnych) oraz OF (ang. *Overflow Flag*, tzw. flaga przepełnienia dla liczb całkowitych) sygnalizują wystąpienie przepełnienia podczas ostatnio wykonanej operacji arytmetycznej.

- W innych przypadkach zaleca się korzystanie z odpowiedniej biblioteki umożliwiającej wykonywanie obliczeń z sygnalizowaniem przepełnień. Przykładem może być stworzona przez firmę Microsoft biblioteka SafeInt, przeznaczona dla języka C++ [2].

- W pewnych sytuacjach najwygodniej jest ograniczyć zakres danych wejściowych do rozsądnych wartości. Dla rozważanego przykładu odpowiednia poprawka mogłaby wyglądać następująco:

```
var szerokosc:uint = PobierzUInt(plik);
var wysokosc:uint = PobierzUInt(plik);
if(szerokosc > 10000 || wysokosc > 10000) {
  throw new Error("Obraz jest zbyt duży.");
}
var wielkosc:int = szerokosc * wysokosc;
```

  Należy jednak zachować szczególną ostrożność przy dobieraniu wartości progowych, aby faktycznie zabezpieczały one przed wystąpieniem przepełnienia. Do istotnych wad tego rozwiązania należy fakt, że testy tego typu mogą nie przetrwać próby czasu (w końcu za 5 lat możemy przesyłać sobie zdjęcia o większych wymiarach).

- Jeśli podejrzewamy, że przepełnienie typu w danym miejscu może nastąpić wskutek przetwarzania poprawnych danych (np. bardzo dużego obrazka), powinniśmy rozważyć rozszerzenie typu zmiennej, np. zmianę 32-bitowego typu *int* na 64-bitowy typ *uint64_t* lub *long long* (dot. popularnych kompilatorów na architekturach x86-32 jak i x86-64). Rozwiązanie to można zastosować również do zabezpieczenia się przed przepełnieniami wskutek przetwarzania specjalnie spreparowanych danych, jeśli jesteśmy w pełni świadomi wartości, jakie mogą przyjmować zmienne podczas wykonywania obliczeń i jesteśmy w stanie dowieść, że rozsze-

rzenie zakresu danej zmiennej faktycznie skutecznie zapobiega wszelkim przepełnieniom.

- W celu wykrycia wystąpienia przepełnienia typu całkowitego w trakcie wykonywania aplikacji napisanej w C lub C++ jako część procesu jej testowania lub debuggowania można użyć opcji *-fsanitize=integer* dostępnej w kompilatorze *clang* w wersji 3.5 i nowszych. Dołączenie owej flagi do opcji kompilacji powoduje, że w momencie wystąpienia przepełnienia na standardowy strumień błędów wypisywana jest informacja o zaistniałej sytuacji, np.:

```
ioc.c:6:13: runtime error: signed integer overflow: 100000 *
100000 cannot be represented in type 'int'
ioc.c:6:17: runtime error: signed integer overflow:
1410065408 * 100000 cannot be represented in type 'int'
```

CIEKAWOSTKA

W wydanej przez MicroProse w roku 1994 grze "Transport Tycoon" można było w prosty sposób doprowadzić do przekroczenia zakresu zmiennej przechowującej ilość posiadanych w grze pieniędzy, dzięki czemu gracz mógł w szybki sposób bardzo się wzbogacić (niestety jedynie wirtualnie).

Błąd objawiał się w momencie zakupu bardzo drogiego tunelu kolejowego, biegnącego przez całą długość mapy. Na przykład, tunel widoczny na zrzucie ekranu z gry poniżej, kosztował ponad 2 miliardy wirtualnych funtów, a konkretniej £2 380 014 463. Niestety, wszystkie operacje finansowe w grze wykonywane były na zmiennych typu *int* (32 bity, liczba ze znakiem), a co za tym idzie, wartość ta nie była mogła zostać wyrażona ponieważ wykraczała poza zakres możliwych wartości - następowało przepełnienie, a wartość była traktowana jak -1 914 952 833 (wynika to z reprezentacji tych liczb na poziomie bitowym).

Oczywiście, gracza posiadającego na początku gry £100 000, według logiki gry, było stać na zakup tunelu kosztującego około minus dwa miliardy, a co za tym idzie, zakup dochodził do skutku. Efektem zakupu jest oczywiście odjęcie kosztu od salda konta, a więc w tym przypadku dochodziło do następujących obliczeń:

```
nowe saldo = 100 000 - (-1 914 952 833) = 1 915 052 833
```

Dzięki temu gracz wzbogacał się o niecałe miliardy wirtualnych funtów i mógł kontynuować rozgrywkę nie martwiąc się o przyszłość swojego przedsiębiorstwa transportowego.

Warto dodać, że błąd został poprawiony w wydanym rok później *Transport Tycoon Deluxe*.

CIEKAWOSTKA

Transport Tycoon Deluxe.

Obcięcie zakresu liczb całkowitych

Kolejny, nie mniej istotny czy rzadziej spotykany rodzaj błędu związany z obliczeniami na liczbach całkowitych to rzutowanie i konwersja zmiennych. Znaczna większość tzw. *silnie typowanych* języków programowania takich jak C, C++, Objective-C, ActionScript czy Java posiada wiele różnych typów zmiennych przeznaczonych do operowania na liczbach całkowitych, które różnią się od siebie przechowywaniem informacji o znaku, zakresem możliwych wartości i co bezpośrednio z tym związane, ilością pamięci, którą taka zmienna zajmuje. W złożonym systemie informatycznym konkretna wartość może być przekazywana przez bardzo długi ciąg zależnych od siebie funkcji, klas i struktur, nierzadko tworzonych w różnym czasie przez zupełnie różne osoby. W związku z tym niezwykle ważne jest, by ostrożnie podchodzić do kwestii konwersji wartości z jednego typu na drugi oraz czynienia jakichkolwiek założeń na temat tego, co może znajdować się w zmiennej otrzymanej z innej części systemu. Niepotrzebna lub nieodpowiednio zaimplementowana konwersja typu może prowadzić do utraty części informacji znajdujących się w zmiennej, co w efekcie nierzadko skutkuje luką bezpieczeństwa w aplikacji czy systemie, lub w gorszym scenariuszu - niepowodzeniem przedsięwzięcia wartego miliardy dolarów.

Wcześniej przyjrzyjmy się jednak poniższemu przykładowemu listingowi kodu w języku C++:

```
int PobierzWartosc() {
  int wartosc;
  scanf("%d", &wartosc);
  return wartosc;
}

void WypiszWartoscChar() {
  char skladnik_1 = PobierzWartosc();
  char skladnik_2 = PobierzWartosc();
  printf("%d\n", skladnik_1 + skladnik_2);
}

void WypiszWartoscInt() {
  int skladnik_1 = PobierzWartosc();
  int skladnik_2 = PobierzWartosc();
  printf("%d\n", skladnik_1 + skladnik_2);
}
```

Pomimo faktu, że obie procedury z rodziny `WypiszWartosc` wykonują tę samą operację, tylko jedna z nich jest w pełni poprawna. Zauważmy, że w przypadku procedury `WypiszWartoscChar` liczba zwrócona przez `PobierzWartosc` znajdująca się w 32 bitowej zmiennej typu `int` zostaje przerzutowana na 8 bitowy typ `char`, co automatycznie wiąże się z utratą górnych 24 bitów danych. Jeśli użytkownik dostarczałby na standardowe wejście wyłącznie liczby z dopuszczalnego zakresu typu `char`, procedura dobrze spełniałaby swoje zadanie; w przeciwnym jednak razie, wyższe bity przekazanych wartości zostałyby zgubione na etapie konwersji, tym samym prowadząc do niewłaściwych wyników na wyjściu.

Niezgodność rozmiaru typów można w wielu przypadkach wykryć na etapie kompilacji; w przypadku C oraz C++, popularne kompilatory z rodziny *gcc* oraz *clang* oferują flagę *-Wconversion*, która aktywuje wypisywanie ostrzeżeń związanych z utratą informacji podczas rzutowania. Ostrzeżenia wygenerowane dla powyższego kodu wygenerowane przez *clang 3.5* wyglądają następująco:

```
test2.c:10:21: warning: implicit conversion loses integer precision:
'int' to 'char' [-Wconversion]
  char skladnik_1 = PobierzWartosc();
       ~~~~~~~~~~   ^~~~~~~~~~~~~~~~
test2.c:11:21: warning: implicit conversion loses integer precision:
'int' to 'char' [-Wconversion]
  char skladnik_2 = PobierzWartosc();
       ~~~~~~~~~~   ^~~~~~~~~~~~~~~~
2 warnings generated.
```

Jeśli kompilator wskaże nam w kodzie miejsca potencjalnie dotknięte problemem utraconej precyzji lub sami takie miejsca zlokalizujemy, nie bójmy się używać typów kompatybilnych z tymi, na których operują używane przez nas moduły aplikacji - potencjalne zaoszczędzenie kilku bajtów nie zrobi w dzisiejszych czasach żadnej różnicy, może zaś uchronić program przed poważnym błędem i jego przyszłymi konsekwencjami.

CIEKAWOSTKA

Choć rzutowanie do zmiennej o zakresie mniejszym niż konieczna do przechowania wartości wydaje się być kwestią trywialną i łatwą do wykrycia, historia pokazuje, że błędy tego typu zdarzały się nawet tam, gdzie w żadnym wypadku nie powinny były się pojawić. Za pierwszy przykład może posłużyć eksplozja jednej z rakiet należących do rodziny Ariane 5. W dniu 4 czerwca 1996, 37 sekund po starcie rakieta warta setki milionów dolarów, wystrzelona przez Europejską Agencję Kosmiczną nakładem siedmiu miliardów dolarów, zboczyła z objętego kursu i rozbiła się na skutek wyjątku spowodowanego nieudaną próbę konwersji wartości zmiennoprzecinkowej zawartej w 64-bitowej zmiennej do 16-bitowej zmiennej typu całkowitego. Więcej informacji na temat katastrofy znaleźć można w licznych internetowych źródłach [3] [4] [5] [6].

Wiele lat później błąd tego samego typu sprawił, że autoryzacja użytkownika przy pomocy hasła stała się bezużyteczna - 9 czerwca 2012 roku Sergei Golubchik opisał na liście dyskusyjnej oss-sec sposób na zalogowanie się do dowolnej podatnej bazy MySQL umożliwiającej zdalne połączenia przy użyciu istniejącej nazwy użytkownika (użytkownik “root” istnieje w znacznej większości baz), bez znajomości poprawnego hasła. Atak polegał na wykonaniu ok. 300 prób logowania do bazy, z których zazwyczaj conajmniej jedna udawała się pomimo dostarczenia błędnych danych autoryzacyjnych. Niewiarygodne? A jednak!

Należy nadmienić, że nie wszystkie kompilacje serwera MySQL zostały dotknięte przez podatność - dotyczyła ona wyłącznie wersji korzystających z funkcji `memcmp` zoptymalizowanej przy użyciu rozszerzenia procesora SSE (Streaming SIMD Extensions), znajdującej się w bibliotece glibc. Sam błąd występował w następującym kodzie:

```
my_bool
check_scramble(const char *scramble_arg, const char *message,
               const uint8 *hash_stage2)
{
  [...]
  return memcmp(hash_stage2, hash_stage2_reassured, SHA1_HASH_SIZE);
}
```

Aby zrozumieć naturę błędu problemu, przyjrzyjmy się deklaracji funkcji standardowej `memcmp`:

```
int memcmp ( const void * ptr1, const void * ptr2, size_t num );
```

Jako, że typ `my_bool` jest zdefiniowany w kodzie źródłowym MySQL jako `char`, potencjalny błąd ukryty w kodzie staje się oczywisty - w funkcji `check_scramble` dochodzi do niejawnej konwersji typów, prowadząc do utraty informacji znajdujących się w górnych 24 bitach wartości zwróconej przez `memcmp`. Zgodnie z dokumentacją, funkcja ta zwraca jako wynik relację pomiędzy porównywanymi blokami pamięci: liczbę ujemną, zero lub dodatnią kolejno kiedy pierwszy region jest leksykograficznie wcześniej, oba regiony są równe i pierwszy region jest leksykograficznie później. Znamy więc znaczenie poszczególnych rodzajów liczb zwracanych przez funkcję - nigdzie nie jest jednak zdefiniowane, *jakie* konkretnie mogą lub powinny być owe ujemne lub dodatnie wyniki. Wiele istniejących implementacji `memcpy` zwraca zwyczajnie -1, 0 lub 1; pozostawienie dowolności co do konkretnej wartości zwracanej pozwoliło jednak na wprowadzenie niskopoziomowych optymalizacji takich jak porównywanie obszarów pamięci w blokach 32-bitowych lub większych, zamiast pojedynczych bajtów. Ponieważ wynik takiej funkcji jest bezpośrednio związany z wielkością jednostki danych na których operuje funkcja, możliwe stają się wartości zwracane takie jak 0x6f203100, które w dalszym ciągu poprawnie (zgodnie ze specyfikacją) sygnalizują, że drugi region pamięci jest mniejszy leksykograficznie. Kiedy jednak liczba ta zostanie skonwertowana do typu `char` tak, jak miało to miejsce w MySQL, zostaje z niej wyłącznie osiem dolnych, zerowych bitów które błędnie sugerują, że ciągi są sobie równe.

W przypadku wspomnianej bazy danych porównywane były hashe SHA1, a ponieważ hash poprawnego hasła nie był znany osobie atakującej, wymagane było przeprowadzenie prostego ataku siłowego ostatecznie doprowadzającego do sytuacji, w której najmniej znaczący bajt wyniku funkcji `memcpy` byłby równy zero. Dzięki właściwościom algorytmu SHA1 prawdopodobieństwo wystąpienia jedynki w każdym z ośmiu bitów wartości zwracanej przez funkcję porównującą było równe 0.5 dla losowego hasła, a więc wykonanie ponad 256 prób logowania z bardzo dużym prawdopodobieństwem umożliwiało poprawne zalogowanie się do systemu.

Do wykorzystania podatności wystarczyła zaledwie jedna linia w Bashu, języku skryptowym powłoki Linuxa, odpowiedzialna za wykonanie tysiąca prób logowania do bazy danych zarządzanej przez podatny serwer:

```
for i in `seq 1 1000`; do mysql -u root --password=bad -h <host>
2>/dev/null; done
```

Błąd ten przeszedł do historii jako jedna z najbardziej absurdalnych luk bezpieczeństwa, wygrywając prestiżową nagrodę w kategorii “Najlepszy błąd po stronie serwera” (ang. *Pwnie for Best Server-Side Bug*). Wkrótce po jego zgłoszeniu wydana została oficjalna łatka, polegająca na objęciu wywołania funkcji `memcmp` makrem o nazwie `test`, *normalizującym* dowolną wartość do zera i jedynki:

```
#define test(a)         ((a) ? 1 : 0)
```

Wycieki danych

Głębokie ukrycie

To nieco humorystyczne określenie opisuje pewną grupę błędów, którą inaczej można by określić jako możliwość znalezienia pewnych danych na stronie WWW (rzadziej w innych technologiach), których położenie z założenia miało być znane jedynie ograniczonej grupie odbiorców, ale z uwagi na pewne niedociągnięcia, dane zostały odnalezione przez inne osoby.

Dobrym przykładem może być część serwisu WWW przedsiębiorstwa, w którym osoba zainteresowana pracą może założyć konto oraz przesłać swoje CV (a w późniejszym terminie je np. uaktualnić). Niestety czasami się zdarza, że kandydat, dysponując linkiem do swojego CV w systemie, może odgadnąć link do CV innych kandydatów. Przykładowy link może wyglądać następująco:

http://example.com/kandydaci/cv/**4713.pdf**

Łatwo się domyślić, że podmieniając liczbę w linku z 4713 na, na przykład, 4712, można prawdopodobnie podejrzeć CV innego kandydata.

CIEKAWOSTKA

Termin "głębokie ukrycie" został spopularyzowany w roku 2010 przez serwis Niebezpiecznik (niebezpiecznik.pl), w artykule o wycieku części bazy dłużników banku PKO BP [7]. Przedstawiciel banku wyjaśniając sytuację miał stwierdzić, że "*plik został zabezpieczony w tak zwanym głębokim ukryciu*". Termin "głębokie ukrycie" został wskazany przez redaktora - "*Macie pomysł, co oznacza "głębokie ukrycie"? ;-)*" - a następnie był używany w innych newsach o podobnej tematyce, np.:

- "7 000 polskich CV w "głębokim ukryciu"" [8]
- ""Głębokie ukrycie" na video" [9]

Ostatecznie termin doczekał się nawet swojej podstrony na polskiej Wikipedii [10]; w kontekście hasła na Wikipedii warto również przeczytać post Pawła Golenia, pt. "O głębokim ukryciu nieco inaczej" [11].

Problem "głębokiego ukrycia" to tak naprawdę kilka innych problemów połączonych razem:

- Po pierwsze - **brak autoryzacji**. Kandydat X nie powinien mieć dostępu do CV kandydata Y - system powinien sprawdzić, czy kandydat X zalogowany na swoje konto ma prawo oglądać CV, które chce wyświetlić (a w przeciwnym wypadku wyświetlić komunikat błędu).
- Po drugie - w takich wypadkach nazwa zasobu nie powinna być po prostu numerem porządkowym (czasami określa się takie przypadki **błędem enumeracji**), ponieważ niepotrzebnie zdradza to liczbę przesłanych CV do danej firmy (co samo w sobie jest wyciekiem danych, chociaż nieznacznym), ale także umożliwia łatwe zgadywanie identyfikatorów CV innych kandydatów. Jednym z rozwiązań tego problemu jest użycie np. sumy SHA-2 z numeru CV oraz pewnego niejawnego ciągu, lub użycie odpowiednio długiego, nieprzewidywalnego, unikatowego identyfikatora.
- Po trzecie - w niektórych przypadkach, nawet jeśli nazwy są losowe, czasami domyślna konfiguracja serwera WWW oferuję listę plików w danym katalogu. Poza szczególnymi przypadkami, zaleca się wyłączenie tej opcji (np. dla serwera Apache należy ustawić Options -Indexes).

Użycie niezainicjalizowanej pamięci

Natywne, niezarządzane języki programowania takie jak C lub C++ charakteryzują się bardzo dużym stopniem zaufania do programisty - to on, i tylko on jest odpowiedzialny za unikanie wszelkich błędów implementacyjnych - zwłaszcza tych związanych z obsługą i dostępem do pamięci. Podejście to ma zarówno wady, jak i zalety: dzięki brakowi narzutu związanego z poprawnością wykonywanych operacji (takich jak dostęp do tablic i dynamicznie zaalokowanych obiektów) poprawnie napisane programy mogą osiągać bardzo dobre wyniki wydajnościowe, w pełni wykorzystując potencjał obliczeniowy sprzętu na którym się wykonują. Z drugiej strony dowolny, nawet prosty błąd stwarza istotne zagrożenie dla bezpieczeństwa, pozwalając potencjalnemu atakującemu na zmianę struktury lub zawartości krytycznych regionów pamięci odpowiadających za poprawny tok wykonania aplikacji i tym samym na przejęcie kontroli nad procesem lub całym systemem.

Obok powszechnie znanych błędów obsługi pamięci związanych z zapisem (np. przepełnienie bufora itd.), które w sposób bezpośredni zagrażają bezpieczeństwu aplikacji, istnieje jeszcze druga grupa - błędy związane z odczytem. Jedną z usterek tego typu jest użycie regionu pamięci, który nie został wcześniej zainicjalizowany ani automatycznie (zgodnie ze standardem, np. w przypadku zmiennych globalnych), ani bezpośrednio w kodzie (w przypadku zmiennych lokalnych lub alokacji dynamicznych).

Błędy użycia niezainicjalizowanej pamięci mogą mieć najróżniejsze objawy i konsekwencje, w zależności od kilku czynników: jakie informacje (semantycznie) przechowuje owa pamięć, w jaki sposób został skompilowany program oraz w jakim środowisku jest on wykonywany (co nierzadko determinuje dane, które przypadkowo znajdują się w niezainicjalizowanych buforach). Istnieje kilka najczęstszych objawów i konsekwencji takich błędów:

- Następuje naruszenie ochrony pamięci, tj. odczyt z niezainicjalizowanej pamięci kończy się niedozwolonym zapisem (pod nieistniejący adres lub poza dozwolone ramy pamięci danego obiektu). Sytuacja taka może wystąpić, jeśli niezainicjalizowana zmienna przechowuje informacje bezpośrednio używane do adresowania pamięci, takie jak wskaźnik, indeks w tablicy lub rozmiar danych w pamięci. Błąd taki jest wtedy często równoznaczny z luką bezpieczeństwa w aplikacji.
- Program nie narusza zasad dotyczących zarządzania pamięcią, jednak zachowuje się w sposób niedeterministyczny, zwracając różne wyniki lub przejawiając różne zachowania w tych samych warunkach i dla tych samych danych wejściowych. Sytuacja taka jest charakterystyczna dla niezainicjalizowanych zmiennych, które przechowują informacje na podstawie których podejmowane są decyzje w aplikacji (np. zmienne typu *bool* używane w wyrażeniach *if*).
- Niezainicjalizowane pozostają wyłącznie regiony pamięci zawierające dane wejściowe. W tym wypadku integralność działania programu nie jest zagrożona (o ile potrafi on poprawnie obsłużyć dowolne dane), jednak wynik działania aplikacji może być niepoprawny, trudny do przewidzenia i najczęściej bezpośrednio związany ze starymi danymi, które znajdowały się w niezainicjalizowanej pamięci.

W sposób oczywisty usterki tego rodzaju naprawia się poprzez inicjalizowanie każdego obszaru pamięci, który jest dalej używany przez program - większym wyzwaniem jest raczej wykrycie ich istnienia. W wielu przypadkach na pomoc może przyjść nam kompilator; spójrzmy na poniższy przykład:

```
  int liczba;
  bool pierwsza;
  bool parzysta;

  scanf("%d", &liczba);
  pierwsza = SprawdzPierwszosc(liczba);
  if (pierwsza && liczba != 2) {
    parzysta = false;
  }

  if (parzysta) {
    puts("Podana liczba jest parzysta.");
  } else {
    puts("Podana liczba nie jest parzysta.");
  }
```

Błąd znajdujący się w kodzie polega na tym, że zmienna `parzysta` wypełniana jest konkretną wartością tylko w przypadku, gdy podana przez użytkownika liczba jest pierwsza; w przeciwnym wypadku pozostanie ona niezdefiniowana (w praktyce na powszechnych platformach przejmując wartość, która wcześniej znajdowała się w tym miejscu na stosie). Jeśli więc na wejściu pojawi się liczba złożona, nie da się z góry przewidzieć wyniku działania programu. Podczas próby skompilowania pliku przy użyciu kompilatora *clang 3.5* z opcją *-Wuninitialized* (wchodzącej w skład opcji *-Wall*), otrzymujemy następujące ostrzeżenia:

```
[...]
uninit.cc:23:7: warning: variable 'parzysta' is used uninitialized
whenever '&&' condition is false [-Wsometimes-uninitialized]
  if (pierwsza && liczba != 2) {
      ^~~~~~~~
uninit.cc:27:7: note: uninitialized use occurs here
  if (parzysta) {
      ^~~~~~~~
uninit.cc:23:7: note: remove the '&&' if its condition is always true
  if (pierwsza && liczba != 2) {
      ^~~~~~~~~~
uninit.cc:19:16: note: initialize the variable 'parzysta' to silence
this warning
  bool parzysta;
               ^
                = false
```

Wskazano nam więc dokładnie, w jakich warunkach flaga nie zostanie zainicjalizowana oraz w którym miejscu jest następnie używana. Śledzenie zależności między zmiennymi na etapie kompilacji jest niestety możliwe wyłącznie w przypadku obiektów statycznych, tj. zdefiniowanych bezpośrednio w kodzie; jeśli zmienimy zmienne `pierwsza` i `parzysta` z lokalnych na dynamicznie alokowane, nie doświadczymy podobnego komunikatu pomimo dalszego istnienia usterki:

```
int liczba;
bool *pierwsza = new bool;
bool *parzysta = new bool;

scanf("%d", &liczba);
*pierwsza = SprawdzPierwszosc(liczba);
if (*pierwsza && liczba != 2) {
  *parzysta = false;
}

if (*parzysta) {
  puts("Podana liczba jest parzysta.");
} else {
  puts("Podana liczba nie jest parzysta.");
}
```

W tej sytuacji możemy jednak polegać na narzędziach o nazwie *memcheck* (wchodzący w skład pakietu *valgrind*) oraz *MemorySanitizer*. Pierwsze z nich jest samodzielną aplikacją, która emuluje wykonywanie dowolnego programu weryfikując przy tym, czy każde z odwołań do pamięci jest poprawne, włączając w to użycie niezainicjalizowanych danych. Przykładowo, uruchomienie przedstawionego wyżej kodu z użyciem *memcheck* skutkuje wypisaniem następującego ostrzeżenia:

```
==14387== Conditional jump or move depends on uninitialised val-
ue(s)
==14387==    at 0x400742: main (uninit.cc:27)
```

O ile bardzo skuteczny i dokładny, *valgrind* posiada jedną, podstawową wadę - wiąże się on z narzutem czasu wykonania rzędu 4-120x, co może być sporą przeszkodą w regularnym testowaniu złożonych aplikacji wykonujących znaczną ilość obliczeń. Znacznie lepiej radzi sobie na tym polu projekt *MemorySanitizer*, który zwalnia typowe aplikacje tylko trzykrotnie dzięki temu, że działa na etapie kompilacji wprowadzając do kodu maszynowego programu dodatkowe instrukcje testujące za każdym razem, czy region do którego następuje odwołanie został wcześniej po-

prawnie zainicjalizowany. *MemorySanitizer* został włączony do kompilatora *clang* w wersji 3.5, a do jego użycia potrzebujemy flagi *-fsanitize=memory*:

```
$ clang++ uninit.cc -fsanitize=memory -g
```

Po uruchomieniu programu i wpisaniu liczby złożonej powinien ukazać nam się komunikat podobny do następującego:

```
==11628== WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7fdcd96ca8a5 in main uninit.cc:27
    #1 0x7fdcd854e76c in __libc_start_main
    #2 0x7fdcd96ca23c in _start

SUMMARY: MemorySanitizer: use-of-uninitialized-value uninit.cc:27
main
Exiting
```

Serdecznie zachęcamy do zapoznania się z oboma systemami detekcji użycia niezainicjalizowanej pamięci, a także wybrania i regularnego korzystania z rozwiązania uznanego za lepsze w konkretnym przypadku.

CIEKAWOSTKA

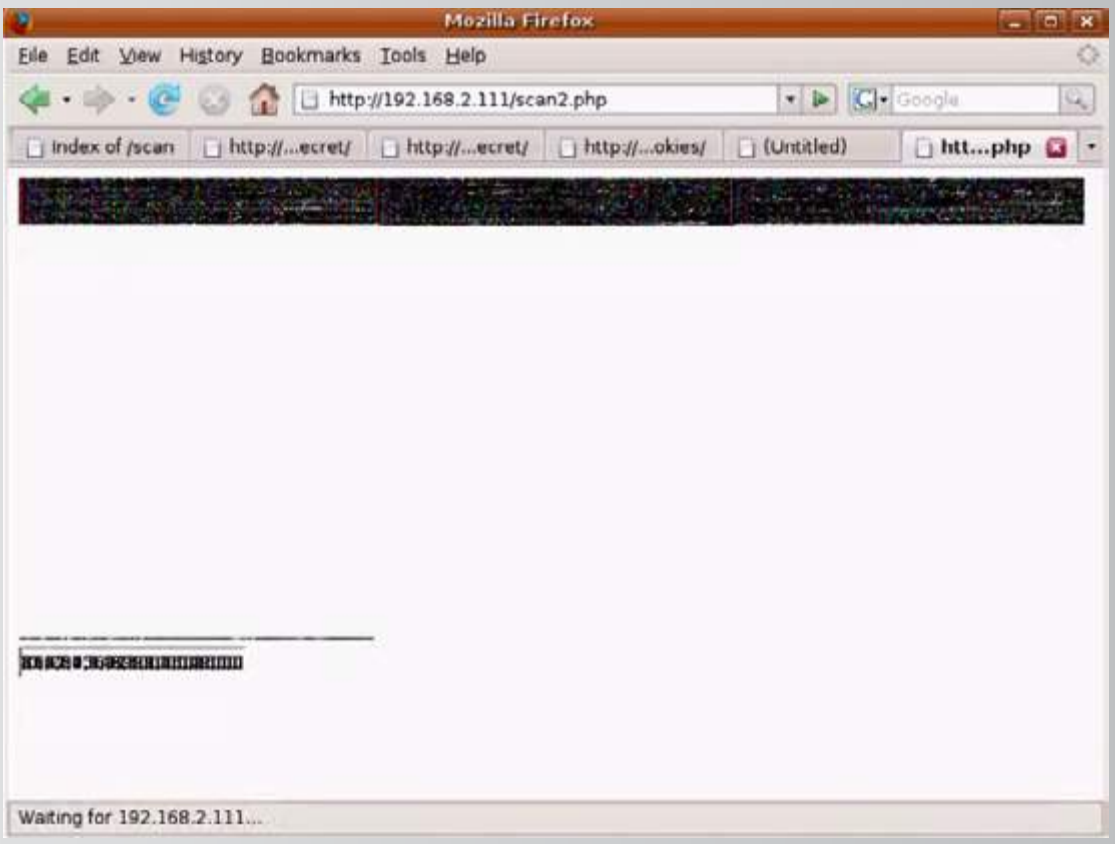
Okazuje się, że błędy użycia niezainicjalizowanej pamięci mogą mieć poważne oraz ciekawe konsekwencje z punktu widzenia prywatności użytkownika. Wyobraźmy sobie następujący scenariusz: użytkownik domowy korzysta z pewnej popularnej aplikacji klienckiej; w pewnym momencie zostaje nakłoniony przez zdalnie atakującą osobę do użycia owego programu do otwarcia pewnego specjalnie spreparowanego pliku. Plik przygotowany przez osobę o niecnych zamiarach wykorzystuje błąd użycia niezainicjalizowanej pamięci w programie, w wyniku czego exploit zyskuje dostęp do "śmieciowych" danych znalezionych w przestrzeni adresowej, a następnie wysyła te dane z powrotem do serwera adwersarza. Brzmi nierealnie? Powyższa sytuacja jest jak najbardziej możliwa jeśli za "pewną aplikację" podstawimy przeglądarkę internetową, plikiem wejściowym będzie plik HTML zawierający sprytny kod Javascript, a błąd będzie polegał na renderowaniu przez przeglądarkę bajtów o niezainicjalizowanej wartości w formie pikseli na stronie.

Zacznijmy jednak od początku - jak dobrze wiemy, wszystkie popularne graficzne przeglądarki internetowe (Internet Explorer, Firefox, Chrome, Opera, Safari) obsługują wiele podstawowych formatów graficznych takich jak BMP, JPEG, GIF czy TIFF. Sześć lat temu autorzy niniejszego artykułu zidentyfikowali niezależnie błędy w obsłudze niektórych z tych formatów (głównie BMP i TIFF) w przeglądarkach Firefox, Opera i Safari. Każdy z nich pozwalał na zmuszenie podatnej przeglądarki do wyświetlenia od 1 do 256 pikseli opisanych przez wartości RGB znajdujące się w niezainicjalizowanej pamięci dynamicznej alokacji, co na pierwszy rzut oka wyglądało jak nic nie znaczące artefakty. Jeśli używana przeglądarka miała wyłączoną obsługę Javascript, na artefaktach się kończyło - obrazek (lub ich większa liczba umieszczona na stronie) wypełniony śmieciami w pamięci wyświetlał się na ekranie, co jednak w żaden nega-

CIEKAWOSTKA

tywny sposób nie wpływało na bezpieczeństwo czy prywatność użytkownika. Kiedy jednak obsługa Javascript była aktywna (co dotyczyło i wciąż dotyczy przeważającej większości użytkowników), odpowiednio spreparowany skrypt był w stanie odczytać wartość renderowanych pikseli przy użyciu dostępnego w HTML elementu *canvas*, a następnie wysłać trzy bajty opisujące kolor każdego z nich do zdalnego serwera. Podsumowując, wyświetlanie niezainicjalizowanej pamięci w kontekście wrogiej strony może prowadzić do ujawnienia fragmentów danych, które wcześniej były przetwarzane przez przeglądarkę. Pozostaje więc pytanie, jakie informacje mogłyby dostać się w niepowołane ręce za sprawą błędu tego typu?

Odpowiedź jest zależna od przeglądarki - w toku empirycznych testów pokazaliśmy, że części kodu HTML stron wyświetlanych w tym samym czasie, urywki komunikacji HTTP i adresy odwiedzonych stron znajdowały się wśród niezainicjalizowanych buforów we wszystkich dotkniętych podatnościami aplikacjach - Firefox, Safari i Opera. Ponadto wśród pamięci wyświetlanej w ramach obrazka w Firefox i Opera znajdowały się wartości ciasteczek należących do zewnętrznych domen, umożliwiając kradzież sesji użytkownika.



Błąd znaleziony przez jednego z autorów niniejszego tekstu (Gynvaela Coldwinda) polegał na nieprawidłowej obsłudze bitmap, które deklarowały istnienie palety o małym rozmiarze (np. jednego koloru), podczas gdy opisy poszczególnych pikseli odwoływały się do wyższych indeksów palety kolorów. Ze względu na fakt, że alokacja odpowiadająca 256 kolorom była inicjalizowana w tym przypadku tylko częściowo, na ekranie była wyświetlana w większości jej stara, niezdefiniowana zawartość. Więcej

informacji na ten temat znaleźć można w oficjalnym dokumencie *advisory* [12].

Z kolei podatność zidentyfikowana przez drugiego z autorów (Mateusza Jurczyka) była związana z obsługą kodowania RLE (*Run-Length Encoding*), prostej metody kompresji stosowanej m.in. w formacie BMP i TIFF. Zachowaniem dekodera sterują zawarte w sekcji danych obrazu "rozkazy"; jedną z możliwych komend jest nakazanie parserowi potraktowanie kolejnych (znajdujących się w pliku) od jednego do 127 bajtów jako dane RGB opisujące kolejne piksele. Jeśli po owej komendzie nie występowała jednak wystarczająca ilość danych (plik kończył się w tym momencie), dekodery błędnie wczytywały do danych obrazu bajty znajdujące się *poza* buforem danych pliku, co w efekcie prowadziło do konsekwencji analogicznych do omówionego wyżej problemu. Zainteresowanych czytelników odsyłamy do lektury dokumentu "*Firefox, Opera, Safari for Windows BMP file handling information leak*" [13].

Obie usterki zostały znalezione i zgłoszone blisko sześć lat temu - nie oznacza to jednak, że poważne błędy użycia niezainicjalizowanej pamięci nie były od tego czasu znajdowane - wręcz przeciwnie, w roku 2013 mogliśmy usłyszeć o wielu ciekawych przypadkach takich problemów - Michał Zalewski opisał podatność w bibliotekach libjpeg6b i libjpeg-turbo (używanej przez większość przeglądarek i aplikacji obsługujących format JPEG) [14], Tavis Ormandy wyjaśnił możliwość podniesienia uprawnień w systemach Windows przy pomocy niezainicjalizowanego wskaźnika dynamicznie alokowanej struktury w graficznym komponencie jądra systemu Windows (win32k.sys) [15], a francuska firma VUPEN wyeksploitowała lukę tej samej klasy w celu przejęcia kontroli nad przeglądarką Internet Explorer [16].

Stabilność i niezawodność

Kolizje w tablicach haszujących

Tablice haszujące (ang. *hash table*) są jedną z najpopularniejszych i najczęściej używanych struktur danych w oprogramowaniu wszelkiego rodzaju. Struktura ta pozwala na tworzenie tablic asocjacyjnych o statystycznie szybkim dostępie do elementów w sytuacjach, w których uniwersum wartości klucza jest znacznie większe od ilości wartości, które przechowywane są w owej tablicy; na przykład, kiedy kluczem mogą być bardzo duże liczby (dziesięcocyfrowe i większe), ciągi tekstowe lub zserializowane obiekty. W najprostszym przypadku tablica haszująca może mieć postać zwykłej tablicy *t* adresowanej liczbami naturalnymi , im większa wartość *n*, tym szybszy dostęp do elementów, ale i większe zużycie pamięci. Aby umożliwić indeksowanie tablicy dowolnymi obiektami, definiuje się tzw. funkcję haszującą *h*, która w sposób deterministyczny odwzorowuje dowolny ciąg bajtów w liczbę z przedziału, zachowując przy tym możliwie równomierny rozkład.

Ponieważ powyższy zakres liczb jest znacznie mniejszy od liczby różnych kluczy tablicy haszującej, funkcja *h* w sposób oczywisty traci znaczną ilość informacji o początkowym obiekcie. Z zasady szufladkowej Dirichleta wynika więc, że w dostępie do tablicy *t* wcześniej czy później wystąpią kolizje, tj. rezultatem funkcji *h* dla dwóch różnych obiektów będzie jednakowa wartość. W celu obsłużenia tego przypadku każda z komórek tablicy zamiast pojedynczej wartości elementu przechowuje listę wszystkich wartości przypisanych do danego hasha. Oznacza to, że jeśli każdy z kluczy użytych podczas wstawiania wartości do tablicy zostanie przetłumaczony na inny indeks tablicy *t*, odczyt dowolnego elementu zajmie czas rzędu O(1). Z drugiej strony, jeśli zdarzy się, że wszystkie elementy znajdą się pod tym samym indeksem, każda z operacji na takiej tablicy może wymagać wykonania kroków w liczbie wprost proporcjonalnej do liczby znajdującej się w niej wartości, podnosząc złożoność pojedynczej operacji do O(n). W większości przypadków, kiedy elementów w tablicy haszującej jest niewiele lub klucze używane do ich zaadresowania dobrane są losowo (a przynajmniej nie mają one żadnych szczególnych właściwości względem funkcji haszującej), przeciętny koszt dostępu nie przekracza zazwyczaj rzędu O(lg n).

Skoro funkcje haszujące zachowują się w sposób deterministyczny a dodatkowo sposób ich działania jest powszechnie znany dla wielu implementacji (w szczególności otwartych projektów), możemy wyobrazić sobie sytuację, w której ktoś specjalnie preparuje wartości kluczy tablicy asocjacyjnej w taki sposób, by użyta funkcja *h* odwzorowała każdą z nich na tę samą pozycję w pomocniczej tablicy *t*. Jeśli kod aplikacji nie spodziewa się wystąpienia tego pesymistycznego przypadku i z góry zakłada, że każda z operacji na tablicy jest szybka w związku z czym wykonuje takich operacji wiele, potencjalnie staje się możliwe znaczne wydłużenie działania podatnej aplikacji. Błąd ten może zostać naprawiony poprzez wprowadzenie pewnego elementu losowości - nie może to być jednak losowy faktor powodujący różnice w wyniku funkcji *h* dla tych samych danych wejściowych w obrębie tej samej sesji programu (gdyż jak pamiętamy, dla poprawności działania tablicy haszującej wymagany jest determinizm funkcji *h*); powinien to być raczej element losowy wprowadzony podczas uruchomienia aplikacji i obowiązujący przez cały czas jej działania.

Przede wszystkim należy jednak pamiętać, że tam, gdzie przetwarzane są dane dostarczane przez użytkownika, wszystko co może pójść nie tak - pójdzie nie tak, gdyż atakujący bardzo chętnie wykorzysta wszystkie dostępne możliwości zaatakowania systemu. Jak się okazuje, zasada ta dotyczy to nie tylko błędów związanych bezpośrednio z jawnie nieprawidłowym przetwarzaniem danych wejściowych (kiedy są one używane w niebezpiecznych operacjach arytmetycznych lub jako część zapytania do bazy danych), lecz może być zastosowana również dla problemów optymalizacyjnych, kiedy aplikacja czyni niekoniecznie prawidłowe założenia na temat natury (formatu, struktury, treści) danych wejściowych.

CIEKAWOSTKA

Problem opisany w tej sekcji okazuje się nie być wyłącznie tworem teoretycznych rozważań - pod koniec 2011 roku na konferencji 28C3 badacze Julian Wälde i Alexander Klink zaprezentowali przykłady praktycznych ataków przeciwko przewidywalności funkcji haszujących zaimplementowanych w technologiach i językach programowania takich jak PHP, Java, .NET, v8, Ruby oraz Python, umożliwiając przeprowadzanie skutecznych ataków typu odmowy usługi (ang. *Denial of Service*) niewielkim kosztem obliczeniowym oraz przepustowości sieciowej, poprzez wymuszenie na serwerze przetwarzania zapytania o złożoności $O(n^2)$, tym samym konsumując nieproporcjonalnie dużą ilość czasu serwera i w przypadku większej liczby takich zapytań wysyłanych na raz - potencjalnie całkowicie uniemożliwiając pracę serwisu internetowego. Problem został uznany za tak poważny i zagrażający ogólnej stabilności sieci, że Microsoft już następnego dnia po publicznym ujawnieniu podatności wydał łatkę naprawiającą błąd w ASP.NET, łamiąc w tym wyjątkowym przypadku regułę zbiorczego publikowania po-

CIEKAWOSTKA

prawek w każdy drugi wtorek miesiąca (dzień znany jako *Patch Tuesday* - w wolnym tłumaczeniu "Łatkowy Wtorek"). Wkrótce za gigantem z Redmond poszły inne projekty oraz producenci, naprawiając błąd poprzez wprowadzenie do implementacji tablic haszujących elementu losowości uniemożliwiającego przewidzenie, jakie ciągi tekstowe spowodują kolizję funkcji haszującej. Zainteresowanych czytelników zapraszamy do obejrzenia nagrania ze wspomnianej wcześniej prelekcji [17], a także zajrzenia do artykułów opisujących rozwój sytuacji w kilka dni po ogłoszeniu informacji o problemie [18] [19].

Wyszukiwanie wszystkiego

Zazwyczaj każdy większy serwis webowy posiada wyszukiwarkę, która w założeniu ma pozwolić szybko znaleźć pożądaną treść w obrębie danej strony. Przykładowo, funkcjonalność ta może być realizowana poprzez wysłanie żądania w formacie podobnym do następującego:

```
http://example.com/szukaj?co=koty&ile=10
```

Takie żądanie może następnie trafić do funkcji, która na jego podstawie wykonuje zapytanie do bazy danych i zwraca wyniki. Np.:

```
# PHP
function Szukaj() {
  global $db;
  $sql = 'SELECT link, text FROM search_table WHERE text LIKE :co LIMIT
:ile';
  $s = $db->prepare($sql);
  $s->bindParam(':co', '%' . $_GET['co'] . '%');
  $s->bind''&''''''Param(':ile', $_GET['ile'], PDO::PARAM_INT);
  $s->execute();
  return $s->fetchAll(PDO::FETCH_ASSOC);
}
```

O ile na pierwszy rzut oka wszystko wygląda poprawnie, o tyle problem zaczyna się, jeśli atakujący wykona zapytanie typu "`?q=%&ile=1000000`", które można przetłumaczyć na "znajdź i wyświetl wszystko" - obciąży to zarówno bazę danych, jak i sam skrypt PHP. Przykładowo, jeśli dany serwis posiada 100 000 podstron, to zwrócone zostanie 100 000 wyników, które muszą najpierw zostać przesłane z bazy danych do silnika PHP, następnie obrobione przez skrypt, a później wysłane do użytkownika. Oczywiście, jedno zapytanie tego typu jest w zasadzie niegroźne, ale należy pamiętać, że atakujący może np. ustawić to zapytanie jako swój awatar na popularnym forum internetowym - spowoduje to, że przeglądarka każdego użytkownika tego forum bezwiednie wyśle powyższe zapytanie w oczekiwaniu na zwrotny obrazek. Duża liczba takich zapytań

może niestety zarówno znacznie obciążyć serwer, jak i łącze na którym stoi serwis.

Prawidłowe podejście do powyższego problemu zakłada:

- Ograniczenie liczby wyświetlanych wyników do sensownej liczby (np. 100).
- Wymaganie, aby tekst którego się szuka miał przynajmniej kilka znaków (zazwyczaj 3).
- Poprzedzenie znaków "%" w zapytaniu odpowiednimi sekwencjami ucieczki (tak, żeby "%" był traktowany po prostu jako znak "%", a nie maska).

Wielowątkowość i wielozadaniowość

Wyścigi w systemie

Jednym z najciekawszych błędów popełnianych podczas tworzenia kodu działającego w wielowątkowym, wielozadaniowym środowisku jest tzw. *TOCTTOU* (czyt. "tok tu"). Błąd ten polega na błędnym założeniu, że współdzielony zasób nie zmieni niespodziewanie swoich właściwości pomiędzy odwołaniami do niego w kontekście tego samego zadania. Nazwa *TOCTTOU* (ang. *Time Of Check To Time Of Use*), którą można dosłownie przetłumaczyć na "moment sprawdzenia kontra moment użycia" wywodzi się z typowego, błędnego schematu kodu:

- W kroku pierwszym następuje sprawdzenie, czy współdzielony zasób jest poprawny (jeśli nie, następuje przerwanie działania).
- W drugim kroku współdzielony zasób zostaje użyty.

Przykładowy kod zawierający opisany wyżej schemat:

```
# Python
def PrzeliczWalute(kwota):
  global kurs

  # Czy znamy aktualny kurs?
  if kurs == 0:
    return False

  DodajDoLogow("PrzeliczWalute", kwota, kurs)

  return kwota / kurs
```

W powyższym kodzie, który mógłby być częścią większego systemu (np. obsługującego kantor internetowy), mamy globalną zmienną kurs, która jest ciągle uaktualniana przez inny, istniejący w systemie wątek, pobierający aktualny kurs waluty z zewnętrznych serwerów. Szczególnym przypadkiem jest błędny kurs równy 0, ustawiany gdy ostatnia próba pobrania kursu zakończyła się błędem (np. w wyniku chwilowego braku łączności).

Kod na pierwszy rzut oka może wydawać się poprawny - najpierw następuje sprawdzenie czy aktualny kurs jest dostępny, a dopiero później dochodzi do

wykonania obliczeń. I tu właśnie tkwi problem - pomiędzy sprawdzeniem, a faktycznym użyciem zmiennej, inny wątek może zmienić jej wartość na 0. W takim wypadku dojdzie do dzielenia przez zero i zostanie rzucony potencjalnie nieprzewidziany wyjątek.

Z TOCTTOU można poradzić sobie na kilka sposobów (w zależności od konkretnej sytuacji):

- Zazwyczaj najlepiej jest synchronizować dostęp do współdzielonego zasobu, tak aby tylko jeden klient na raz mógł na nim operować.
- W niektórych przypadkach (jeśli ograniczamy użycie zasobu do jego odczytów) wystarczy wykonać lokalną kopię wartości danego zasobu i operować jedynie na kopii - dzięki temu zapobiegamy niespodziewanej zmianie właśności.

CIEKAWOSTKA

Jednym z problemów występujących na dzielonych hostingach oferujących PHP był dostęp do plików, których właścicielem nie był właściciel skryptu PHP. Wynika to m.in. z używanej czasami konfiguracji, w której silnik PHP działa z uprawnieniami serwera WWW i jako taki musi mieć dostęp do plików stron WWW (wraz ze skryptami PHP) wszystkich użytkowników na danym hostingu.

Jednym z (nienajlepszych) rozwiązań jest użycie tzw. funkcjonalności `open_basedir` w konfiguracji PHP - jest to opcja określająca, do jakich katalogów silnik PHP ma zezwalać na dostęp - opcje tę można ustawić np. dla konkretnej domeny lub konkretnego użytkownika. W takim przypadku, jeśli skrypt próbuje otworzyć pewien plik następuje najpierw sprawdzenie, czy ów plik leży w katalogu znajdującym się na liście `open_basedir`.

Fragment przykładowej konfiguracji serwera Apache z PHP oraz opcją `open_basedir` może wyglądać następująco:

```
<VirtualHost *:80>
  ...
  ServerName alice.example.com
  php_admin_value open_basedir /home/alice/alice.example.com/
  <Directory /home/alice/alice.example.com/>
    ...
    allow from all
  </Directory>
  ...
</VirtualHost>

<VirtualHost *:80>
  ...
  ServerName mallory.example.com
```

CIEKAWOSTKA

```
  php_admin_value open_basedir /home/mallory/mallory.example.
com/
  <Directory /home/mallory/mallory.example.com/>
    ...
    allow from all
  </Directory>
  ...
</VirtualHost>
```

W październiku 2006 roku znany badacz Stefan Esser znalazł prostą lukę typu *TOCTTOU*, pozwalającą ominąć restrykcje związane z `open_basedir` [20]. Błąd wynikał z prostego faktu, że pomiędzy sprawdzeniem, czy dana ścieżka prowadzi do pliku (lub katalogu), który leży w drzewie wymienionym w `open_basedir`, a faktycznym otwarciem danego pliku, mija trochę czasu. W związku z tym, jeśli ścieżka w danym momencie wskazywałaby na zasób do którego skrypt powinien mieć dostęp, tym samym przechodząc test, a następnie zostałaby zmieniona (korzystając z mechanizmu linków symbolicznych) by wskazywać na np. plik innego użytkownika, ostatecznie otwarty zostałby właśnie plik innego użytkownika.

Badacz zaproponował, że można osiągnąć to uruchamiając jednocześnie dwa skrypty PHP. Pierwszy próbowałby (aż do skutku) odczytać zawartość pliku innego użytkownika, np. “xxx/home/alice/tajne.dane”. Drugi natomiast składałby się z następującego kodu PHP:

```
mkdir("a/a/a/a/a/a");
while(1) {
  symlink("a/a/a/a/a/a", "dummy");
  symlink("dummy/../../../../../../", "xxx");
  unlink("dummy");
  symlink(".", "dummy");
}
```

W przypadku wygranego wyścigu (tj. gdy podmiana linku symbolicznego nastąpi w odpowiednim momencie) test `open_basedir` widziałby próbę dostępu do pliku “a/a/a/a/a/a/../../../../../../../home/alice/tajne.dane”, czyli po prostu “./home/alice/tajne.dane” - a to jest jak najbardziej zgodne z `open_basedir`. Kilka kroków później otwarty zostałby plik “./../../../../../../home/alice/tajne.dane” - czyli plik “tajne.dane” w katalogu domowym użytkownika Alice.

Co ciekawe, błąd ten był o tyle specyficzny, że jego rozwiązaniem była sugestia wyłączenia funkcji *symlink* w konfiguracji PHP [21], oraz stwierdzenie, że `open_basedir` nigdy nie miał być barierą między użytkownikami działającymi na tym samym serwerze [22].

Dane wejściowe

Deserializacja niezaufanych danych

Tworząc serwisy WWW w niektórych sytuacjach wygodnie jest wymieniać z przeglądarką dane w postaci zserializowanej. Dzięki takiemu rozwiązaniu po stronie serwera wystarczy dane zdeserializować i od razu można zacząć ich używać, bez wcześniejszej konwersji typów, tworzenia tablic czy obiektów - wszystko to ma miejsce automatycznie podczas procesu deserializacji. W takich przypadkach oczywiście skrypt po stronie klienta albo musi odpowiednio zserializować dane, albo przekazać serwerowi zserializowane dane, które otrzymał od niego "na przechowanie" poprzednio (mogą to być np. wybrane preferencje użytkownika).

Przykładowy kod realizujący odbiór danych:

```
# Python
def OdbierzDane(parametry):
  if "obiekt" not in parametry:
    return False
  return pickle.loads(parametry["obiekt"])

// PHP
function OdbierzDane() {
  if(!isset($_POST["obiekt"]))
    return false;
  return unserialize($_POST["obiekt"]);
}
```

Metody serializacji można podzielić na proste, które obsługują jedynie podstawowe typy i struktury (np. JSON), oraz bardziej złożone, które mogą przechowywać również obiekty różnych klas (np. użyte wyżej Pickle i PHP serialize/unserialize). Serializacja i deserializacja obiektów to proces bardziej skomplikowany - wynika z faktu, że obiekt to nie tylko zbiór wartości, ale często również różne dodatkowe meta-informacje, które nie wynikają bezpośrednio z wartości pól obiektu (takie jak np. aktywne połączenie sieciowe, czy uchwyt otwartego pliku). Z tego też powodu przy deserializacji obiektu nie wystarczy odtworzyć wartości zmiennych; trzeba również obiekt "obudzić" - czyli np. otworzyć odpowiednie pliki, wznowić połączenia sieciowe itp.

Ponieważ niemożliwym jest aby kod funkcji deserializującej umiał "obudzić" obiekty wszystkich istniejących klas, zadanie to spada na samą klasę, a konkretniej, na specjalne metody, które są wywoływane podczas deserializacji. Nazwy oraz budowa metod zależne są oczywiście od konkretnego serializera; przykładowo, podczas deserializacji:

Pickle wywołuje metodę `__setstate__` dla danego obiektu (oczywiście, o ile takowa istnieje).
PHP unserialize wywołuje metodę `__wakeup`.

Gdyby spojrzeć na ten problem z perspektywy deserializacji niezaufanych danych, okazuje się, że skoro potencjalny atakujący może kontrolować typy danych w zserializowanym pakiecie, to może on również spowodować wywołanie metod "budzących" dany obiekt na danych kontrolowanych przez siebie. Co więcej, skoro przy deserializacji dany obiekt zostaje na nowo stworzony, to w pewnym momencie działania skryptu zostanie on również zniszczony, a wtedy wywołany zostanie oczywiście destruktor (dla naszych przykładowych języków jest to: metoda `__del__` w Pythonie, oraz `__destruct` w PHP), który również będzie operował na wartościach pól wybranych przez atakującego. Czy takie sprowokowanie wywołania funkcji "budzących" oraz destruktorów powoduje poważniejsze problemy? Okazuje się, że tak.

CIEKAWOSTKA

W lutym 2013 Egidio Romano opublikował informację o błędzie związanym z deserializacją niezaufanych danych w popularnym silniku CMS Joomla! [23]. Badacz wskazał destruktor klasy *plgSystemDebug*, który w pewnym momencie wykonywał następujący kod:

```
$filterGroups = (array) $this->params->get('filter_groups',
null);
```

Z uwagi na to, że atakujący kontrolował również typ pola *params* w obiekcie, mógł on doprowadzić do wywołania metody *get* w dowolnej innej istniejącej klasie (oczywiście pod warunkiem, że metoda *get* w danej klasie istniała). Badacz wskazał dwie możliwości przeprowadzenia ataku dalej:

- Metodę *get* w klasie *JInput*, która wywołuje metodę *clean* na innym kontrolowanym obiekcie. Metodę *clean* można było znaleźć w klasie *JCacheStorageFile* i pozwalała ona na usunięcie dowolnego katalogu, wraz z zawartością, co mogło spowodować utratę danych.
- Jeszcze ciekawsza metoda *get* została odnaleziona w klasie *JCategories* - doprowadzenie do jej wywołania pozwalało atakującemu na wykonanie częściowo kontrolowanego zapytania do bazy SQL, a w konsekwencji do wycieku danych z bazy.

O ile w przypadku PHP powaga potencjalnych konsekwencji deserializacji niezaufanego ciągu w znacznym stopniu zależy od tego, jakie klasy są obecne w aplikacji, o tyle w przypadku języka Python oraz biblioteki *Pickle* sytuacja przedstawia się znacznie gorzej. Wynika to z metody działania *Pickle* - zserializowane dane są w zasadzie mini-programem, który ma na celu odbudowanie danych do ich pierwotnej postaci (tj. tej sprzed serializacji). Podczas deserializacji niewielki interpreter wykonuje ów mini-program, a jego rezultatem są listy, obiekty itp. Oczywiście w takim wypadku konsekwencje zależą głównie od możliwości dopuszczanych przez interpreter - a te, jak się okazuje, są znaczące z uwagi na instrukcję "R". W tym miejscu warto wspomóc się źródłami interpretera Pythona (Python 2.7.3, *Lib/pickle.py*):

```
REDUCE          = 'R'   # apply callable to argtuple, both on stack
...
    def load_reduce(self):
        stack = self.stack
        args = stack.pop()
        func = stack[-1]
        value = func(*args)
        stack[-1] = value
    dispatch[REDUCE] = load_reduce
```

Jak można wywnioskować z powyższego kodu, instrukcja "R" ściąga ze stosu maszyny wirtualnej funkcję do wywołania (nazwę funkcji oraz modułu można umieścić na stosie za pomocą instrukcji "c"), oraz jej argumentu, po czym ją wywołuje. W rezultacie deserializacja niezaufanych danych może doprowadzić do wykonania dowolnego kodu przez atakującego, a w konsekwencji do przejęcia przez niego kontroli nad aplikacją.

CIEKAWOSTKA

Deserializacja niezaufanych danych nie jest domeną tylko i wyłącznie aplikacji webowych. W roku 2011 Marco Slaviero zgłosił błąd w systemie serwerowym Red Hat dzięki któremu, właśnie za pomocą spreparowanych danych, był wstanie podnieść uprawnienia swojego użytkownika do pełnych uprawnień konta root [24].

Błąd znajdował się w *backendzie* graficznego interfejsu służącego do zarządzania firewallem - *system-config-firewall*, a konkretniej w skrypcie *system-config-firewall-mechanism.py*, który działał z uprawnieniami root. Podatny kod wyglądał następująco [25]:

```
@dbus.service.method(DBUS_DOMAIN, in_signature='s', out_signature='i')
def write(self, rep):
  try:
    args = pickle.loads(rep.encode('utf-8'))
  except:
     return -1
```

W powyższym kodzie argument wyeksportowanej przez DBUS metody `write` był bez-

CIEKAWOSTKA

pośrednio poddawany deserializacji. Ponieważ dowolny użytkownik systemu miał prawa (korzystając z DBUS) do wywołania metody `write`, mógł również doprowadzić do wykonania zdefiniowanego przez siebie kodu (np. nadającego mu pełne uprawnienia administratora).

Błąd został naprawiony poprzez zastąpienie *Pickle* dużo prostszym formatem *JSON*.

Podsumowując - w kodzie, w którym zależy nam na bezpieczeństwie, powinno unikać się wykonywania "złożonych" deserializacji. Na przykład, zamiast *Pickle* w Pythonie czy *serialize*/*unserialize* w PHP możemy użyć formatu JSON (który jest dużo prostszy i sam w sobie nie oferuje serializacji złożonych obiektów) co automatycznie eliminuje wszystkie opisane wyżej problemy.

W przypadku w którym klient podaje nam jedynie zserializowane dane, które wcześniej otrzymał od serwera, można zastosować HMAC w celu upewnienia się, że dane nie zostały zmienione przez użytkownika.

Logika aplikacji

Oscylatory walutowe

Jednym z ciekawszych błędów logiki aplikacji bankowych są błędy zaokrąglenia, czasem nazywane oscylatorami walutowymi. Przykładowy schemat takiego oscylatora wygląda następująco (na potrzeby przykładu załóżmy, że kurs EUR/PLN wynosi 4.00):

1. Atakujący zakłada dwa konta w banku: walutowe (w EUR) oraz w natywnej walucie (PLN).
2. Przelewa na konto PLN niewielką sumę pieniędzy (wystarczy kilka groszy).
3. Przelewa 3 grosze z konta w PLN na konto walutowe w EUR.
4. (jeśli błąd występuje) Na koncie EUR pojawia się 1 eurocent (0.01 EUR) z uwagi na zaokrąglenie w górę.
5. Następnie atakujący przelewa 0.01 EUR z konta walutowego na konto w PLN.
6. (jeśli błąd występuje) Na koncie w PLN pojawiają się 4 grosze (a więc o 33% więcej).
7. Punkty 3-6 są powtarzane w nieskończoność, a atakujący za każdym razem zyskuje jeden grosz).

Działanie powyższego oscylatora można utrudnić na kilka sposobów:

- Wprowadzając niewielką, stałą, opłatę od wymiany walut, jeśli transakcja jest poniżej pewnej kwoty.
- Blokując możliwość robienia niewielkich przelewów.
- Można również zliczać różnicę wynikającą z zaokrąglenia i ją wyrównywać (a więc jeśli w danej transakcji klient zyskał np. 1 grosz, to byłyby on odejmowany przy następnej transakcji w drugą stronę).

CIEKAWOSTKA

Bardzo dobry opis oscylatorów walutowych bazujących na błędzie zaokrąglenia przedstawił, na odbywającej się w Paryżu konferencji Nuit du Hack 2013 (a później na odbywającej się w Moskwie konferencji ZeroNights 2013), badacz Adrian Furtuna [26] [27]. Oprócz obszernego wytłumaczenia tematu, badacz zaprezentował również własnej

CIEKAWOSTKA

produkcji urządzenie pozwalające zautomatyzować dokonywanie przelewów wymagających potwierdzenia tokenem. Urządzenie wprowadzało kod transakcji w tokenie, po czym odczytywało go za pomocą kamery sprzężonej z OCR; następnie kod mógł zostać użyty do zatwierdzenia przelewu.

Fotografia: *Adrian Furtunã, "Practical exploitation of rounding vulnerabilities in internet banking applications"*

Według przedstawionych wyliczeń, urządzenie pozwalało na wykonanie do 14400 zautomatyzowanych transakcji dziennie, co w przypadku opisanym przez badacza, mogło generować do 68 USD (korzystając tylko z jednego konta).

CIEKAWOSTKA

Warto również przeczytać artykuł "EURO page: Conversion Arithmetics" autorstwa Keesa Vuika [28], w którym autor wyjaśnia problemy związane z zaokrągleniem przy przejściu danego kraju z lokalnej waluty na EURO.

Dziwactwa

Nierówne równości

Jednym z problemów występujących w językach o słabym typowaniu jest porównywanie zmiennych o różnych typach - wymaga to zazwyczaj konwersji danych, a nie zawsze jest oczywiste w jaki sposób to zrobić. Czasami dochodzi więc do sytuacji, które mogą zaskoczyć niejednego programistę - a co za tym idzie, doprowadzić do zaistnienia błędu. Z tego powodu języki programowania tego typu posiadają często dwa różne operatory porównania - tzw. "operator równości wartości" (zazwyczaj zapisywany jako ==), oraz "operator równości wartości i typu" (zazwyczaj zapisywany jako ===). Przykładami takich języków mogą być JavaScript oraz PHP.

Poniżej znajduje się kilka przykładów pochodzących z języka PHP - wszystkie poniższe porównania dają w wyniku wartość *true*, a więc porównane wartości są uznawane za równe [29]:

- `"1000" == "0x3e8"`
- `"1234" == "  \t\r\n 1234"`
- `"123451234512345123451234512345" == "123451234512345123451234512346"` (dla wersji 5.4.3 oraz starszych)
- `5 == "5 tysiecy"`
- `9223372036854775807 == "9223372036854775808"` (dla 64-bitowych wersji PHP)
- `0 == "ala ma kota"`
- `fopen('plik', 'w') == "0.002e3"` (pod warunkiem, że *fopen* zwróci zasób o ID równym #2)

Najlepszym sposobem na uniknięcie nieprzewidzianych problemów związanych z porównaniami jest konsekwentne stosowanie operatora *silnego* porównania (===), z wyjątkiem konkretnych miejsc, w których *słabe* porównanie działa dokładnie tak, jak tego oczekujemy.

CIEKAWOSTKA

W sierpniu roku 2012 Arseny Reutov, badacz z Positive Research Center, znalazł poważną lukę w popularnym silniku forum internetowego *Simple Machines Forum*, pozwalającą na przejęcie konta dowolnego użytkownika, w tym administratora. Błąd znajdował się w kodzie, którego zadaniem była weryfikacja wygenerowanego wcześniej kodu pozwalającego zresetować zapomniane hasło do konta. Kod korzystał z operatora słabej równości (a raczej nierówności w tym wypadku) i wyglądał następująco:

```
if (/* ... */ || substr($realCode, 0, 10) != substr(md5($_POST['code']),
0, 10))
  // ... kod się nie zgadza; pokaż błąd ...
```

Powyższy kod sprawdzał, czy pierwsze dziesięć znaków kodu jest równe pierwszym dziesięciu znakom sumy MD5 z podanego przez użytkownika (atakującego) kodu. W tym przypadku celem atakującego było wylosowanie takiego $realCode (poprzez wielokrotne próby odzyskania hasła - a więc wielokrotne re-generowania $realCode), żeby zaczynał się on od znaków “0e” lub “1e”, a po nich następowały jedynie cyfry dziesiętne, np.:

`0e71264128` lub `1e95723510`

Oczywiście, atakujący nie znał poprawnego kodu, ale i nie musiał go znać. Należy zwrócić uwagę, że z punktu widzenia PHP oraz “słabego” porównania, powyższe wartości to jedynie liczby typu float w notacji naukowej, które są równe 0 (w końcu $0 * 10^{71264128}$ to po prostu 0) lub 1. Wystarczało więc, aby atakujący w pole *code* wpisał dowolny ciąg, którego pierwsze dziesięć znaków sumy MD5 również miało formę “0e<cyfry>” lub “1e<cyfry>”.

Na przykład, dla podanej wyżej wartości (“0e71264128”) wystarczyło podać kod “astronavigation”, ponieważ, pierwsze dziesięć znaków sumy MD5 tego słowa to “0e86666438” (czyli również wartość 0).

```
> php -a
Interactive shell


php > var_dump("0e71264128" == substr(md5("astronavigation"), 0, 10));
bool(true)
```

Błąd został naprawiony w lutym ubiegłego roku [30].

Kryptografia

Wartości (nie)losowe

Podczas tworzenia serwisów webowych oraz podobnych aplikacji działających po stronie serwera, czasem zachodzi konieczność wygenerowania losowego, trudnego do zgadnięcia ciągu znaków. Taki ciąg może zostać użyty jako np.:

- Identyfikator sesji użytkownika.
- Początkowe hasło do konta użytkownika (ustalane przy tworzeniu konta).
- Kod pozwalający zresetować hasło użytkownika.
- CAPTCHA.
- Unikatowa nazwa pliku lub podobnego rodzaju identyfikator zasobu.
- Token anti-XSRF [31].

Przykładowy, błędny kod generujący losowy ciąg w PHP może wyglądać następująco:

```
function WygenerujToken($dlugosc) {
  $token = "";
  $alfabet = "abcdefghijklmnopqrstuvwxyz";
  for($i = 0; $i < $dlugosc; $i++)
    $token .= $alfabet[mt_rand(0,25)];

  return $token;
}
```

Problem w powyższym kodzie polega na użyciu tzw. kryptograficznie niebezpiecznego generatora liczb losowych (w tym wypadku jest to `mt_rand` korzystający z algorytmu *Mersenne Twister* [32]). Atakujący dysponujący jednym lub kilkoma tokenami może spróbować zrekonstruować wewnętrzny stan generatora, a co za tym idzie, odtworzyć wygenerowane wcześniej oraz/lub później tokeny (na współczesnym komputerze jest to w przypadku *Mersenne Twister* kwestia kilku sekund).

Do generowania losowych ciągów, na których ma opierać się bezpieczeństwo systemu, powinno używać się tzw. kryptograficznie bezpiecznych generatorów. Większość technologii udostępnia programistom odpowiednie mechanizmy pozwalające na wygenerowanie bezpiecznych liczb losowych, na przykład:

- PHP: `openssl_random_pseudo_bytes`
- Python: `os.urandom`
- Java: `java.security.SecureRandom`
- Ruby: `SecureRandom`
- Windows: `CryptoGenRandom`
- GNU/Linux: pseudo-urządzenia `/dev/random` lub `/dev/urandom`

CIEKAWOSTKA

Niestety zdarza się, że błędy pojawiają się również w funkcjach, które z założenia miały generować kryptograficznie bezpieczne wartości.

W roku 2008 znaleziono błąd w generatorze liczb losowych biblioteki OpenSSL dystrybuowanej z systemami z rodziny Debian [33]. Okazało się, że z uwagi na niewielkie zmiany w kodzie entropia generatora liczb losowych spadła do 16-bitów, przez co np. OpenSSH generujący klucz SSH RSA był w stanie wygenerować jedynie 65536 różnych, przewidywalnych par kluczy. W takim wypadku wystarczyło aby atakujący wygenerował wszystkie możliwe klucze u siebie, a następnie mógł uzyskać dostęp do konta użytkownika, który miał ustawioną autoryzacje na jeden z wadliwych kluczy.

Inny poważny przypadek odkryty w zeszłym roku dotyczył platformy Android. W tym wypadku generator liczb losowych biblioteki OpenSSL był nieprawidłowo inicjalizowany, przez co generowane klucze stały się przewidywalne [34]. Niestety, podobno doprowadziło to do kradzieży pewnej sumy kryptowaluty Bitcoin przechowywanej na portfelach, do których klucze zostały wygenerowane wadliwym generatorem.

Przenośność kodu

Nazwy plików

Przenoszenie kodu między różnymi platformami jest z jednej strony łatwe - cała logika pozostaje niezmieniona - ale z drugiej strony natrafia się na drobne różnice sprawiające, że kod zachowuje się inaczej na różnych systemach. Jedną z oczywistych różnic między systemami operacyjnymi jest kwestia systemu plików, a w szczególności nazw plików. Chodzi tutaj zarówno o proste kwestie jak np. czy dany znak może wystąpić w nazwie pliku lub katalogu, jak i bardziej skomplikowane, takie jak dodatkowe możliwości niektórych systemów plików.

Poniższa tabela zawiera kilka przykładowych różnic pomiędzy typowymi systemami plików używanymi na systemach GNU/Linux (np. ext3) oraz Windows (NTFS):

Windows	GNU/Linux
Separatorem nazw katalogów i plików może być zarówno znak “/” (slash) jak i “\” (backslash).	Separatorem nazw katalogów i plików jest znak “/” (slash).
Znaku < > oraz \ nie mogą wystąpić w nazwie pliku lub katalogu.	Znaku < > oraz \ mogą wystąpić w nazwie pliku lub katalogu.
Do pliku “XYZ” można odwołać się używając nazwy “xyz”, jak i “xYz”, itd. Co więcej, można też skorzystać z nazwy “xyz.....”, “xyz “, jak i (w przypadku NTFS) “xyz::$DATA”.	Pliki “XYZ” oraz “xyz” to dwa różne pliki.
Maksymalna długość ścieżki to 32767 znaków.	Brak limitu długości ścieżki.
Tylko administrator może tworzyć linki symboliczne.	Dowolny użytkownik może tworzyć linki symboliczne.

Powyższe różnice stają się niesamowicie istotne w przypadku programów, które muszą podjąć decyzję bazując na nazwie pliku (lub ścieżce) otrzymanej od zewnętrznego użytkownika. Przykładem może być poniższy kod:

```
def CzyMoznaWyswietlicPlik(nazwa):
  return nazwa.find("config") == -1
```

Powyższy kod ma decydować, czy żądany plik może zostać wyświetlony użytkownikowi - w tym przypadku jedynym ograniczeniem jest zabronienie wyświetlenia dowolnego pliku, który ma w nazwie (pliku lub ścieżki) słowo "config". Oczywiście, o ile funkcja ta będzie poprawnie działać na systemach z rodziny GNU/Linux, to w przypadku Windowsa użytkownik może po prostu zażądać pliku "CoNfIg" i to wystarczy aby obejść ten filtr.

W poprawnym kodzie zamiast zabraniać dostępu do konkretnych plików, pozwolibyśmy na dostęp jedynie do niektórych, uniemożliwiając dostęp do wszystkich innych. W innym wypadku należałoby zaproponować oddzielną logikę dla różnych systemów operacyjnych.

CIEKAWOSTKA

Na błąd tego typu natrafili twórcy serwera WWW Lighttpd. Okazało się, że o ile odwołanie typu "http://example.com/index.php" działało zgodnie z założeniami (tj. skrypt PHP był wykonywany), o tyle "http://example.com/index.php::$DATA" powodowało wyświetlenie źródeł skryptu PHP [35] - wynikało to z zasady działania alternatywnych strumieni (ADS) w systemie plików NTFS. Ostatecznie błąd nie został poprawiony, pojawiła się jedynie sugestia, aby na platformie Windows nie zezwalać na pojawienie się dwukropka w nazwie zasobu. Warto nadmienić, że podobne problemy wystąpiły w roku 1998 w serwerze Microsoft IIS [36].

Zakończenie

Jak pokazaliśmy w artykule, błędy towarzyszyły programistom już dziesiątki lat temu, i towarzyszą im wciąż po dziś dzień, zmieniając jedynie naturę oraz technologie zgodnie z najnowszymi trendami w światku rozwoju oprogramowania. Biorąc pod uwagę olbrzymią liczbę ludzi zajmujących się zawodowo programowaniem oraz zwyczajną ludzką omylność, można pokusić się o stwierdzenie, że wszystkie możliwe pomyłki w kodzie zostały już popełnione - niektóre z nich miały jednak zaskakujące objawy oraz konsekwencje, nierzadko czyniąc systemy informatyczne podatnymi na różnego rodzaju ataki zagrażające bezpieczeństwu danych użytkowników lub powodując inne, znaczące straty. Pamiętajmy więc, by starannie pisać kod starając się przewidzieć wszelkie przypadki brzegowe, w żadnym wypadku nie ufać danym pochodzącym od (potencjalnie wrogiego) użytkownika, a przede wszystkim wszechstronnie i wyczerpująco testować i audytować rozwijane oprogramowanie - w przeciwnym wypadku ryzykujemy znalezieniem się w kolejnym opracowaniu najciekawszych błędów w historii *software*, czego sobie ani Czytelnikowi oczywiście nie życzymy.

AUTORZY

Mateusz (po lewej) od wielu lat pasjonuje się tematyką bezpieczeństwa komputerowego - specjalizuje się w metodach odnajdowania oraz wykorzystywania podatności w popularnych aplikacjach klienckich oraz systemach operacyjnych. Na codzień pracuje w firmie Google na stanowisku Information Security Engineer, w wolnych chwilach prowadzi bloga związanego z bezpieczeństwem niskopoziomowym (*http://j00ru.vexillium.org*).

Gynvael na co dzień 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.

Bibliografia

[1] Goudey H. (2013). Chrome on a Nexus 4 and Samsung Galaxy S4 falls. http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/Chrome-on-a-Nexus-4-and-Samsung-Galaxy-S4-falls/ba-p/6268679
[2] Microsoft Corporation. SafeInt. http://safeint.codeplex.com/
[3] Kees Vuik. Some disasters caused by numerical errors. http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html
[4] James Gleick. A Bug and a Crash (Sometimes a Bug Is More Than a Nuisance). http://www.around.com/ariane.html
[5] Wikipedia. Cluster (spacecraft). http://en.wikipedia.org/wiki/Cluster_(spacecraft)
[6] Jean-Marc Jezequel (IRISA), Bertrand Meyer (ISE). Design by Contract: The Lessons of Ariane. http://archive.eiffel.com/doc/manuals/technology/contract/ariane/
[7] Niebezpiecznik. "Głębokie ukrycie" danych w PKO BP. http://niebezpiecznik.pl/post/glebokie-ukrycie-danych-w-pko-bp/
[8] Niebezpiecznik. 7 000 polskich CV w "głębokim ukryciu". http://niebezpiecznik.pl/post/7-000-cv-w-glebokim-ukryciu/
[9] Niebezpiecznik. "Głębokie ukrycie" na video. http://niebezpiecznik.pl/post/glebokie-ukrycie-na-video
[10] Wikipedia. Głębokie ukrycie. http://pl.wikipedia.org/wiki/G%C5%82%C4%99bokie_ukrycie
[11] Paweł Goleń. O głębokim ukryciu nieco inaczej. http://wampir.mroczna-zaloga.org/archives/1084-o-glebokim-ukryciu-nieco-inaczej.html
[12] Gynvael Coldwind. FireFox 2.0.0.11 and Opera 9.50 beta Remote Memory Information Leak. http://vexillium.org/?sec-ff
[13] Mateusz "j00ru" Jurczyk. Firefox, Opera, Safari for Windows BMP file handling information leak. http://gynvael.vexillium.org/ext/vc/shadow_002/mateusz_j00ru_jurczyk_memory_leak_2008.pdf
[14] Michal Zalewski. Bugs in IJG jpeg6b & libjpeg-turbo. http://seclists.org/fulldisclosure/2013/Nov/83
[15] Tavis Ormandy. Introduction to Windows Kernel Security Research. http://blog.cmpxchg8b.com/2013/05/introduction-to-windows-kernel-security.html
[16] Nicolas Joly (VUPEN). Advanced Exploitation of IE MSXML Remote Uninitialized Memory (MS12-043 / CVE-2012-1889). http://www.vupen.com/blog/20120717.Advanced_Exploitation_of_Internet_Explorer_XML_CVE-2012-1889_MS12-043.php
[17] Alexander "alech" Klink, Julian "zeri" Wälde. Efficient Denial of Service Attacks on Web Application Platforms. http://www.youtube.com/watch?v=R2Cq3CLI6H8
[18] Mike Lennon (SECURITYWEEK). Hash Table Vulnerability Enables Wide-Scale DDoS Attacks. http://www.securityweek.com/hash-table-collision-attacks-could-trigger-ddos-massive-scale
[19] Jon Brodkin (Ars Technica). Huge portions of the Web vulnerable to hashing denial-of-service attack. http://arstechnica.com/business/2011/12/huge-portions-of-web-vulnerable-to-hashing-denial-of-service-attack/
[20] Stefan Esser. Advisory 08/2006: PHP open_basedir Race Condition Vulnerability. http://www.hardened-php.net/advisory_082006.132.html

[21] Mandriva. MDKSA-2006:185.
http://www.mandriva.com/en/support/security/advisories/advisory/MDKSA-2006:185/?name=MDKSA-2006:185
[22] Red Hat Bugzilla. Bug 169857.
http://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=169857#c1
[23] Egidio Romano. Analysis of the Joomla PHP Object Injection Vulnerability.
http://karmainsecurity.com/analysis-of-the-joomla-php-object-injection-vulnerability
[24] Red Hat Bugzilla. Bug 717985.
https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2011-2520
[25] Thomas Woerner. (Patch) Replace pickle by json.
https://bugzilla.redhat.com/attachment.cgi?id=511508
[26] Adrian Furtuna. Practical exploitation of rounding vulnerabilities in internet banking applications.
http://www.youtube.com/watch?v=t4Er3Jwhr6Y
[27] Adrian Furtuna. Practical exploitation of rounding vulnerabilities in internet banking applications.
http://2013.zeronights.org/includes/docs/Adrian_Furtuna_-_Practical_exploitation_of_rounding_vulnerabilities_in_internet_banking_applications.pdf
[28] Kees Vuik. Some disasters caused by numerical errors - EURO page: Conversion Arithmetics.
http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html#euro
[29] Gynvael Coldwind. PHP equal operator ==. http://gynvael.coldwind.pl/?id=492
[30] Security Lab. PT-2012-29: Administrator Privilege Gaining in Simple Machines Forum.
http://en.securitylab.ru/lab/PT-2012-29
[31] Wikipedia. Cross-site request forgery.
http://pl.wikipedia.org/wiki/Cross-site_request_forgery#Tw.C3.B3rcy_stron
[32] Wikipedia. Mersenne twister. http://en.wikipedia.org/wiki/Mersenne_twister
[33] Debian (wiki). SSLkeys. https://wiki.debian.org/SSLkeys
[34] Alex Klyubin. Some SecureRandom Thoughts.
http://android-developers.blogspot.com.au/2013/08/some-securerandom-thoughts.html
[35] Lighttpd. Bug #1335. http://redmine.lighttpd.net/issues/1335
[36] Microsoft. Microsoft Security Bulletin MS98-003 - File Access Issue with Windows NT Internet Information Server (IIS). http://technet.microsoft.com/en-us/security/bulletin/ms98-003

IBUK libra

Czytelnia czynna całą dobę!

PWN

Czytaj, gdzie chcesz i jak chcesz!

Z KAŻDEGO MIEJSCA:

- *dom*
- *biblioteka*
- *uczelnia*
- *praca*

NA KAŻDYM URZĄDZENIU:

- *komputer*
- *laptop*
- *tablet*
- *smartfon*

24/7 POGOTOWIE CZYTELNICZE

CHCESZ SKORZYSTAĆ Z PLATFORMY IBUK LIBRA?

Sprawdź czy Twoja biblioteka (instytucja) ma wykupiony dostęp do wybranych publikacji!

www.libra.ibuk.pl

CHCESZ ZAPEWNIĆ DOSTĘP SWOIM CZYTELNIKOM, WSPÓŁPRACOWNIKOM, KLIENTOM?

Skontaktuj się z nami: kontakt@libra.ibuk.pl

PZWL

CZASOPISMA MEDYCZNE
WYDAWNICTWA LEKARSKIEGO

DOŁĄCZ DO GRONA
NASZYCH PRENUMERATORÓW

PZWL

PZWL Wydawnictwo Lekarskie Sp. z o. o.
Grupa PWN
ul. G. Daimlera 2
02-460 Warszawa

prenumeraty@pzwl.pl
tel. 502 796 171
czasopisma.pzwl.pl

WYDAJ WSZYSTKICH SWOICH BOHATERÓW

rozpisani.pl

Projektujemy. Edytujemy. Drukujemy. Wydajemy.
Dystrybuujemy i sprzedajemy Twoją książkę.

Ty zajmij się tym, co kochasz. PISZ!

Więcej na: info@rozpisani.pl | fb.com/rozpisani

rozpisani·pl