Hartowanie kodu

W ostatnich latach podatności bezpieczeństwa komputerowego zaczęły trafiać na pierwsze strony gazet. Czy to poprzez chwytliwie brzmiące nazwy typu Heartbleed lub Spectre, czy to w wyniku ogromnych wycieków danych organizacji rządowych lub mediów społecznościowych. Nawet największe i najbardziej szanowane firmy z branży IT, które mogą sobie pozwolić na własne zespoły specjalistów ds. bezpieczeństwa komputerowego, nie są w stanie poradzić sobie z prostym faktem – nowoczesne oprogramowanie staje się tak duże i tak skomplikowane, że błędy bezpieczeństwa wydają się nieuchronne.

W tym artykule przyjrzymy się podejściu do projektowania oprogramowania, które pomaga eliminować całe klasy błędów, bez potrzeby zatrudniania setek inżynierów bezpieczeństwa, którzy będą zajmować się stałym audytem kodu.

Podejście to w firmie Google znane jest od lat pod szyldem *safe coding*. Wzmianki o nim można znaleźć m.in. w artykule Christopha Kerna pt. *Securing the Tangled Web*[1] w *Communications of the ACM (2014)*.

CZEMU TRUDNO UNIKNĄĆ BŁĘDÓW BEZPIECZEŃSTWA?

Popularne języki programowania i biblioteki były przez lata projektowane z myślą o ekspresywności, nie o bezpieczeństwie, jako nadrzędnym celu. Aby napisać bezpieczny kod, często trzeba się mocno nagimnastykować i mieć dużą wiedzę, a i tak często musimy polegać na tym, jak zachowuje się reszta oprogramowania. To ostatnie zmienia się z biegiem czasu, często w innych plikach źródłowych, a zmiany te dokonywane są przez różnych członków zespołu. Założenia, które zapewniały kontrolowany dostęp do wrażliwych danych, mogą się zmienić w sposób kompletnie niezauważalny.

Jak pokazuje doświadczenie, nawet najmniejsze błędy zostaną skrupulatnie wykorzystane przez kreatywnych atakujących.

JAK WYGENEROWAĆ URL

Na przykładzie tworzenia adresów URL pokażemy, jak można nie tylko ułatwić audyt kodu, ale też i całkowicie wyeliminować taką potrzebę w większości przypadków.

Listing 1. Generowanie adresu URL

```
String url = "https://" + address;
if (isCustomer(user)) {
  List<String> urlParams = new ArrayList<String>();
  for (Map.Entry<String, String> p: params) {
    urlParams.add(
      p.getKey() + "="
      + URLEncoder.encode(p.getValue(), StandardCharsets.UTF_8));
  }
  url += "/?" + Strings.join("&", urlParams);
}
```

Kod w Listingu 1 generuje adres URL. W linijce 7 da się zauważyć, że autor próbuje w jakiś sposób zabezpieczyć swój program poprzez kodowanie wartości parametru zapytania. Mimo tego jest w nim wiele potencjalnych zagrożeń. Główne z nich to:

1. Klucze słownika `params` nie są kodowane przed dodaniem ich do `urlParams`. Czy jest to pomyłka, czy może są już one w poprawnej formie w słowniku?[2]
2. Nie wiemy niczego o tym, co może zawierać zmienna `address`. Czy jest pod kontrolą użytkownika? Czy może się to zmienić z biegiem czasu w otaczającym kodzie?

BEZPIECZEŃSTWO KOSZTEM ELASTYCZNOŚCI

Powyższy sposób na budowanie adresów URL daje nam bardzo dużo swobody. Taka elastyczność z pewnością zostanie wykorzystana do granic przez kreatywnych programistów. Nawet jeśli mają oni najlepsze intencje, znacząco podnosi to prawdopodobieństwo pojawienia się błędu – jeśli nie od razu, to po wielu modyfikacjach kodu z biegiem czasu.

W powyższym przykładzie znajduje się kilka błędów. Jednak nawet gdyby one nie występowały, bardzo trudno jest taki kod zrecenzować (ang. *code review*). Zmienna `address` na pierwszy rzut oka może przyjąć dowolne wartości, niewiele wiadomo o tym, czy klucze w słowniku `params` są odpowiednio zakodowane itd.

Zamiast swobodnego sklejania napisów, proponujemy funkcję `TrustedURL::FromConstant`, której prototyp wygląda tak:

```
TrustedURL FromConstant(StringLiteral url)
```

gdzie `StringLiteral` jest typem, którego wartościami są wyłącznie literały napisowe (różnie realizowane w zależności od języka programowania, patrz ramka „Akceptowanie wyłącznie stałych czasu kompilacji”).

1. https://dl.acm.org/doi/abs/10.1145/2643134

2. Niestety, aplikowanie `URLEncoder.encode` „na wszelki wypadek” nie zda tutaj egzaminu, bowiem podwójne zakodowanie popsułoby adres URL.

Akceptowanie wyłącznie stałych czasu kompilacji

Poniżej przedstawiamy dwie techniki na wymuszanie użycia stałych czasu kompilacji przez bezpieczne API.

Java: @CompileTimeConstant

Error Prone – narzędzie do statycznej analizy kodu Javy – pozwala na wykorzystanie adnotacji `@CompileTimeConstant` (https://errorprone.info/bugpattern/CompileTimeConstant).

Go: prywatny typ string

```
package trustedurl

type stringLiteral string

func FromConstant(url stringLiteral) TrustedURL {
  return TrustedURL{url: url}
}
// ...
```

Ponieważ typ `stringLiteral` nie jest wyeksportowany przez paczkę `trustedurl`, nie można jawnie stworzyć żadnej wartości tego typu. Niemniej jednak literał napisowy przekazany do `FromConstant` zostanie zaakceptowany przez kompilator.

Działać ona będzie tylko dla bardzo prostych przypadków, w których cały URL jest znany w momencie pisania kodu. Możemy jednak dodać metodę:

```
TrustedURL
  WithParams(StringLiteral url,
  Map<StringLiteral, String> queryParams)
```

która pozwoli na przekazywanie parametrów zapytania. `WithParams` automatycznie podda te parametry odpowiedniemu kodowaniu.

Zagwarantowaliśmy w taki sposób dwie własności `TrustedURL`:

1. Protokół, adres serwera i ścieżka są pod pełną kontrolą programisty. Poprzez wymóg użycia literałów napisowych liczba możliwych adresów jest znacząco ograniczona i łatwiejsza do audytu przez inżyniera bezpieczeństwa. Mamy pewność, że adres serwera został wyjęty spod potencjalnej kontroli użytkownika (atakującego).
2. Jedyna część URL, która jest pod kontrolą użytkownika, to parametry. Te kodowane są w poprawny sposób przez funkcję `WithParams`.

W kontekście używania adresów URL możemy uznać zatem wszystkie wartości typu `TrustedURL` za **bezpieczne.**

Z wykorzystaniem `TrustedURL` nasz kod wygląda tak:

Listing 2. Generowanie adresu URL z wykorzystaniem TrustedURL

```
TrustedURL url;
if (isCustomer(user)) {
  url = TrustedURL.WithParams(
    "https://customer.example.com", params);
} else {
  url = TrustedURL.FromConstant("https://shop.example.com");
}
```

Oczywiście, jest szansa, że programista **umyślnie** przekaże jako pierwszy argument niebezpieczny adres – jednak będzie to widoczne gołym okiem podczas recenzji kodu.

Ograniczenia w językach programowania

Od dawna w świecie programowania korzysta się z ograniczeń dotyczących tego, co programiście wolno, a czego nie, aby zagwarantować pewne zachowanie programów. Dla przykładu, większość języków z *odzyskiwaniem pamięci* (ang. *garbage collection*) nie pozwala na arytmetykę wskaźników, eliminując tym samym ogromną klasę błędów, z którymi borykał się chyba każdy programista C.

Innym przykładem z ostatnich lat jest język Rust, który zdobywa coraz większą popularność. Otworzył on oczy wielu programistom na pojęcie *pożyczania* (ang. *borrowing*), które pozwala na wyeliminowanie błędów związanych m.in. z *wiszącymi referencjami* (ang. *dangling reference*), bez korzystania z dynamicznego zarządzania pamięcią.

TYLKO JEDNA DROGA DO CELU…

Aby w pełni wykorzystać zalety płynące z bezpiecznych API, należy sprawić, aby były one jedynym sposobem na osiągnięcie celu.

Wszystkie biblioteki, które korzystają z parametrów będących adresami URL, powinny zostać zmienione tak, aby akceptowały je wyłącznie jako `TrustedURL`.

Taint checking

Taint checking wydaje się na pierwszy rzut oka podobną techniką do *bezpiecznych wartości* takich jak `TrustedURL`. Główna różnica polega tutaj jednak na tym, że w *bezpiecznym stylu* zaczynamy od punktu braku zaufania (tzn. każda wartość jest potencjalnie niebezpieczna), zamiast polegać na tym, że programiście uda się *skazić* (ang. *taint*) wszystkie niebezpieczne wartości.

…Z KILKOMA WYJĄTKAMI

Z pewnością czytelnicy mogą sobie wyobrazić, że serwis nasz nie jest zawarty tylko w jednej bibliotece, w jednym języku programowania. Co zrobić w sytuacji, kiedy pytamy nasz własny backend o adres URL? Wynikiem zapytania nie będzie stała czasu kompilacji, więc nie możemy użyć ani `TrustedURL::FromConstant`, ani `TrustedURL::WithParams`. Jak poradzić sobie w takiej sytuacji?

ReviewedConversions

Wprowadzamy paczkę `ReviewedConversions`, która zawiera metodę:

```
TrustedURL FromSecurityReviewedString(const String s)
```

Jedyne, co ona robi, to opakowuje podany argument w `TrustedURL` **bez sanityzacji, kodowania czy weryfikacji.**

Paczka ta powinna być chroniona (np. poprzez widoczność (https://docs.bazel.build/versions/master/visibility.html) lub lintery), tak aby każde jej użycie **wymagało recenzji przez zespół inżynierów bezpieczeństwa**. Audyt taki powinien być przeprowadzony zgodnie z pewnymi zasadami, nakreślonymi poniżej.

Zasada 1: Tylko gdy absolutnie konieczne

Ponieważ każde użycie paczki `ReviewedConversions` powinno przejść przez specjalistyczną recenzję, powinno się jej używać, tylko gdy jest to absolutnie konieczne.

Użycie `ReviewedConversions::FromSecurityReviewedString` **samo w sobie nie podnosi bezpieczeństwa oprogramowania**. Paczka ta pozwala jedynie na:

- sygnalizację, że dany fragment kodu niesie ze sobą pewne ryzyko,
- inwentarz i analizę podatności,
- możliwość integracji `TrustedURL` w pozostałej części projektu, gdzie bezpieczeństwo jest zapewnione poprzez `FromConstant` oraz `WithParams`.

Zasada 2: Ogranicz kontekst i zależności

Wywołanie `FromSecurityReviewedString` powinno być maksymalnie odizolowane od reszty kodu. Im bardziej odporny na czynniki zewnętrzne będzie URL, który przekażemy do opakowania jako `TrustedURL`, tym lepiej. Gdy to możliwe, nie polegajmy na argumentach funkcji czy innych dynamicznych wartościach.

Przykład: jeśli często pytamy nasz backend o URL, stwórzmy oddzielną funkcję, wewnątrz której wysyłamy zapytanie do zaufanego serwera i zwracamy `TrustedURL`. W ten sposób zapewniamy, że żaden kod nie będzie w stanie wpłynąć (a przez to zwiększyć ryzyko) na wartość przekazywaną do `FromSecurityReviewedString`.

Zasada 3: Zapewnij bezpieczeństwo użycia

W Zasadzie 1 zostało wspomniane, że użycie `ReviewedConversions` samo w sobie nie podnosi bezpieczeństwa oprogramowania. Każde użycie powinno być poprzedzone sprawdzeniami (weryfikacją), które pozwalają:

- zrecenzować kod w łatwy sposób,
- ograniczyć wpływ otaczającego kodu na bezpieczeństwo użycia `ReviewedConversions`.

Przykład

```
// URL ma zawsze bardzo prostą, bezpieczną postać.
// Trzy do pięciu liter na początku, a potem ".foo.com".
verify(regexp.Match("https://[a-z]{3-5}.foo.com", url))
return ReviewedConversions.FromSecurityReviewedString(url)
```

Nie należy pozwalać na użycie `ReviewedConversions`, kiedy bezpieczeństwo nie jest w żaden sposób zagwarantowane przez kod.

INTEGRACJA Z ISTNIEJĄCYM OPROGRAMOWANIEM

Wiemy już, jak radzić sobie z nowym kodem, który chce korzystać z bezpiecznych adresów URL. Co jednak zrobić, kiedy nasz projekt istnieje już od kilku lat i nie możemy sobie teraz pozwolić na poświęcenie tygodni na refaktoryzację istniejącego kodu za jednym zamachem?

LegacyConversions

Podobnie do `ReviewedConversions`, tworzymy paczkę `LegacyConversions` z funkcją `ToTrustedURL`. W przeciwieństwie jednak do tej pierwszej **nie będziemy pozwalać na pojawianie się nowych jej użyć**. `ReviewedConversions` służy wyłącznie do oznaczenia istniejących miejsc w kodzie, które powinny być w przyszłości poddane refaktoryzacji.

Takie podejście pozwala na wprowadzanie zmian w sposób iteracyjny, stopniowo zwiększając bezpieczeństwo istniejącego już oprogramowania. Dobrym pomysłem jest zobowiązanie się do refaktoryzacji jednego użycia `LegacyConversions` na członka zespołu w danym cyklu rozwoju oprogramowania (np. w sprincie).

Biblioteki wymuszające użycie TrustedURL

Po oznaczeniu wszystkich istniejących adresów URL za pomocą `LegacyConversions` zmieniamy nasze biblioteki tak, aby akceptowały jedynie `TrustedURL`. Każda nowa linijka kodu musi korzystać z `FromConstant`, `WithParams` lub, w skrajnych przypadkach, z `ReviewedConversions`. Te ostatnie powinny być każdorazowo recenzowane przez inżynierów bezpieczeństwa, zgodnie ze wspomnianymi poprzednio zasadami.

Przykłady bezpiecznych API

Odpowiednikiem przedstawionego tutaj `TrustedURL` w bibliotece Closure (JavaScript) jest `goog.html.TrustedResourceUrl`[1]. Korzystając z systemu szablonów **Closure Templates**[2], możemy zagwarantować, że wszystkie adresy URL serwowane przez naszą stronę będą tworzone w bezpieczny sposób.

Mechanizm **Trusted Types** pozwala na ograniczanie modyfikacji DOM, m.in. poprzez przypisania do `innerHTML`. Więcej na https://web.dev/trusted-types/.

1. https://google.github.io/closure-library/api/goog.html.TrustedResourceUrl.html
2. https://github.com/google/closure-templates

PODSUMOWANIE

Adaptując *bezpieczny styl*, sprawiamy, że bezpieczeństwo naszego projektu jest możliwe do utrzymania na dużą skalę i na długi czas. Zmniejszamy zapotrzebowanie na ciągły audyt kodu przez specjalistów ds. bezpieczeństwa kosztem elastyczności.

Dodatkowo, poprzez użycie `ReviewedConversions` możemy mieć na oku potencjalnie ryzykowny kod, a `LegacyConversions` pozwala nam stopniowo podnosić bezpieczeństwo projektu bez potrzeby wielkiej, jednorazowej refaktoryzacji.

Gdzie stosować?

Nie zawsze jest oczywiste, gdzie można zastosować zasady programowania w bezpiecznym stylu. Istnieje jednak kilka obszarów, które występują w większości oprogramowania, dla których warto rozważyć adaptację powyższych technik:

- systemy szablonów HTML czy korzystanie z DOM API (aby zapobiec m.in. atakom XSS),
- bazy danych SQL (aby zapobiec atakom SQL injection),
- kontrola dostępu (aby mozna było generować domyślnie bezpieczne konfiguracje).

DAMIAN BOGEL

@kele_codes | kele@google.com

Pracuje w Google nad bezpiecznym kodem. Zaangażowany w różne projekty wewnątrz firmy związane z Go. Członek redakcji https://pagedout.institute. Fan komiksu „Strange Planet".