OBRONA I ANALIZY ŚLEDCZE

bootkitów, rootkitów i innych zagrożeń dla BIOS-u

PWNING_WEEK 2021

PWN

17

JAK DZIAŁA UEFI SECURE BOOT

W poprzednich rozdziałach omawialiśmy wprowadzenie zasady podpisywania kodu trybu jądra, co skłoniło twórców złośliwego oprogramowania do przejścia od stosowania rootkitów w stronę bootkitów, przenosząc wektor ataku z jądra systemu operacyjnego na niechronione komponenty rozruchowe. Tego rodzaju oprogramowanie uruchamiane jest przed załadowaniem systemu operacyjnego, zatem jest w stanie ominąć lub wyłączyć mechanizmy zabezpieczeń systemu. Oznacza to, że aby wymusić zabezpieczenia i zagwarantować bezpieczeństwo, system operacyjny musi być w stanie wykonać rozruch przez zaufane środowisko, którego składniki nie zostały zniekształcone.

W tym miejscu do gry wkracza technologia UEFI Secure Boot, temat tego rozdziału. Mająca na celu przede wszystkim ochronę komponentów rozruchowych przed modyfikacjami i zagwarantowanie, że tylko zaufane

moduły są ładowane i wykonywane podczas rozruchu, UEFI Secure Boot może być skutecznym rozwiązaniem dla ochrony przed zagrożeniami stwarzanymi przez bootkity – o ile pokryje wszystkie kierunki ataku.

Niemniej jednak ochrona proponowana przez UEFI Secure Boot jest podatna na działanie rootkitów oprogramowania układowego (*firmware*), najnowszej i najszybciej rosnącej technologii złośliwego oprogramowania. W rezultacie potrzebujemy kolejnej warstwy zabezpieczeń, aby objąć nimi cały proces rozruchowy, od samego początku. Można to osiągnąć przez implementację Secure Boot, nazywanego *Verified Boot* oraz *Measured Boot* (*zweryfikowany* oraz *mierzony rozruch*).

W tym rozdziale przedstawimy kluczowe elementy tej technologii zabezpieczeń. Najpierw opiszemy, jak, po umocowaniu w sprzęcie, pozwala chronić przed rootkitami oprogramowania, a następnie omówimy szczegóły implementacji i sposoby ochrony potencjalnych ofiar przed bootkitami.

Jednak, jak to często bywa w branży bezpieczeństwa, bardzo nieliczne rozwiązania zabezpieczeń mogą zapewnić ostateczną ochronę przed atakami; napastnicy i obrońcy są związani wiecznym wyścigiem zbrojeń. Rozdział ten zakończymy omówieniem słabości UEFI Secure Boot, sposobami jego ominięcia i metodami ochrony przy użyciu dwóch wersji weryfikowanego lub mierzonego rozruchu, oferowanych przez Intel i ARM.

Czym jest Secure Boot?

Główny cel Secure Boot to powstrzymanie kogokolwiek przed wykonaniem nieautoryzowanego kodu w środowisku przeduruchomieniowym; inaczej mówiąc, tylko kod spełniający zasadę integralności platformy może zostać wykonany. Ta technologia jest bardzo ważna dla platform wysokiej niezawodności, ale jest również często stosowana w urządzeniach wbudowanych i platformach mobilnych, gdyż pozwala dostawcom ograniczyć platformę do używania zaakceptowanego przez nich oprogramowania, jak w przypadku iOS dla iPhone lub systemu operacyjnego Windows 10 S.

Secure Boot występuje w trzech odmianach, zależnych od poziomu w hierarchii procesu rozruchu, na którym jest wymuszany:

Secure Boot dla systemu operacyjnego – implementowany na poziomie programu ładującego system operacyjny, weryfikuje komponenty ładowane przez ten program ładujący, takie jak jądro systemu i sterowniki rozruchowe;

Secure Boot dla UEFI – implementowany w oprogramowaniu układowym UEFI; weryfikuje sterowniki i aplikacje DXE, układy Option ROM oraz programy ładujące system operacyjny;

Secure Boot dla platformy (weryfikowany albo mierzony Secure Boot) – zamocowany w sprzęcie; weryfikuje oprogramowanie układowe inicjujące platformę.

Secure Boot dla systemu operacyjnego omówiliśmy w rozdziale 6, zatem w tym rozdziale skupimy się na rozwiązaniach UEFI Secure Boot oraz weryfikowanym lub mierzonym rozruchu.

Szczegóły implementacji UEFI Secure Boot

Rozpoczniemy to omówienie od tego, jak działa UEFI Secure Boot. Po pierwsze, trzeba zauważyć, że UEFI Secure Boot jest częścią specyfikacji UEFI, którą można znaleźć pod adresem *http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf*. Będziemy odwoływać się do tej specyfikacji – innymi słowy, do opisu tego, jak UEFI Secure Boot *powinno* działać – choć różni wytwórcy platform mogą mieć odmienne szczegóły implementacji.

UWAGA *W tym podrozdziale, począwszy od tego miejsca, ilekroć odwołujemy się do „Secure Boot", mamy na myśli UEFI Secure Boot, chyba że zostanie wskazane inaczej.*

Zaczniemy od przyjrzenia się sekwencji rozruchu, aby zobaczyć, w którym momencie Secure Boot wchodzi do gry. Następnie pokażemy, jak Secure Boot uwierzytelnia pliki wykonywalne i omówimy uczestniczące w tym bazy danych.

Sekwencja rozruchu

Przypomnijmy pokrótce sekwencję rozruchową UEFI opisaną w rozdziale 14, aby zobaczyć, w którym miejscu do procesu tego wkracza Secure Boot. Jeśli ktoś pominął ten rozdział, warto do niego teraz powrócić.

Jeśli powrócimy do podrozdziału „Jak działa oprogramowanie układowe UEFI" na stronie 269, zobaczymy, że pierwszym fragmentem wykonywanym, gdy system wychodzi z resetu, jest oprogramowanie układowe inicjowania platformy, wykonujące podstawowe inicjowanie sprzętu. W trakcie inicjowania platformy chipset i kontroler pamięci są nadal w stanie niezainicjowanym: żadna pamięć DRAM nie jest jeszcze dostępna dla oprogramowania układowego, a urządzenia peryferyjne na magistrali PCIe nie zostały jeszcze wyliczone. (Magistrala PCIe to standard magistrali szeregowej dużej prędkości, używany w praktycznie wszystkich współczesnych komputerach; omówimy ją bliżej w dalszych rozdziałach). W tym momencie Secure Boot nie jest jeszcze aktywne, co oznacza, że część inicjowania platformy systemowego oprogramowania układowego nie jest chroniona w tym punkcie.

Gdy oprogramowanie inicjowania platformy odkryje i skonfiguruje RAM i wykona podstawowe inicjowanie platformy, przechodzi do załadowania sterowników DXE i aplikacji UEFI, które z kolei kontynuują inicjowanie sprzętu platformy. To w tym momencie do gry wkracza Secure Boot.

Zakotwiczony w oprogramowaniu układowym inicjowania platformy, Secure Boot wykorzystywany jest do uwierzytelnienia modułów UEFI ładowanych z układu flash SPI (Serial Peripheral Interface) lub układów Option ROM urządzeń peryferyjnych.

Mechanizm uwierzytelniania używany w Secure Boot to zasadniczo proces weryfikacji podpisu cyfrowego. Tylko poprawnie uwierzytelnione obrazy są dopuszczone do wykonania. Do zarządzania kluczami weryfikacji podpisów Secure Boot wykorzystuje infrastrukturę klucza publicznego (*public key infrastructure* – PKI).

Wyjaśniając prosto, implementacja Secure Boot zawiera klucz publiczny służący do weryfikowania cyfrowych podpisów obrazów wykonywalnych ładowanych podczas rozruchu. Obrazy te powinny mieć osadzony podpis cyfrowy, choć, jak zobaczymy dalej w tym rozdziale, istnieją pewne wyjątki od tej reguły. Jeśli obraz przejdzie weryfikację, jest ładowany i ostatecznie wykonywany. Jeśli obraz nie ma podpisu i weryfikacja się nie powiedzie, wywołuje to zachowanie remediacyjne – działania wykonywane w sytuacji, gdy Secure Boot zawiedzie. W zależności od zasad system może kontynuować rozruch normalnie albo przerwać proces rozruchu i wyświetlić użytkownikowi komunikat błędu.

Rzeczywiste implementacje Secure Boot są nieco bardziej skomplikowane niż to, co tu opisaliśmy. Aby poprawnie ustanowić zaufanie w kodzie wykonywanym podczas rozruchu, Secure Boot używa różnych typów baz danych podpisów, kluczy i zasad. Przyjrzyjmy się tym czynnikom i ich szczegółom jeden po drugim.

Praktyczne implementacje: kompromisy

Producenci rzeczywistych implementacji oprogramowania układowego UEFI często idą na kompromis między zabezpieczeniami a wydajnością. Sprawdzanie podpisu cyfrowego każdego obrazu UEFI żądającego wykonania zajmuje czas. We przeciętnej współczesnej platformie może występować kilkaset obrazów UEFI próbujących się załadować, zatem weryfikacja podpisu cyfrowego każdego pliku wykonywalnego wydłużyłoby proces rozruchu. Jednocześnie producenci są pod presją na zredukowanie czasu rozruchu, szczególnie w przypadku systemów wbudowanych i w branży motoryzacyjnej. Zamiast weryfikowania każdego obrazu UEFI dostawcy oprogramowania układowego często wybierają weryfikowanie obrazów UEFI przy użyciu skrótów, aby poprawić wydajność. Zbiór skrótów dla dozwolonych obrazów jest umieszczony w rozwiązaniu pamięciowym, którego integralność i autentyczność jest sprawdzana tylko raz, przez podpis cyfrowy, przy dostępie do tej pamięci. Te skróty omówimy bardziej szczegółowo w dalszej części rozdziału.

PWNING_WEEK 2021

Dołącz do uczestników PWNing Week 2021!

Grupa PWN zaprasza do udziału w **PWNing Week 2021** – jednym z ważniejszych w Polsce wydarzeń z obszaru cyberbezpieczeństwa. Tegoroczna edycja odbędzie się online i jest rozszerzoną formułą odbywającej się w ubiegłych latach konferencji **Security PWNing**.

Dowiedz się więcej

Tegoroczną edycję kierujemy szczególnie do kadry zarządzającej, osób odpowiedzialnych za wybór i zakup rozwiązań z zakresu bezpieczeństwa i monitoringu sieci oraz poszukujących nowych rozwiązań chmurowych.

PWNing WEEK 2021 to:

- Security Conference (17-19 listopada)
- strefa EXPO (17-19 listopada)
- warsztaty, szkolenia i inne wydarzenia towarzyszące (15-16 listopada, 22-26 listopada)

Tematyka PWNing Security Conference:

- Bezpieczeństwo danych w dobie pandemii i masowej pracy zdalnej
- Rozwiązania chmurowe
- Narzędzia wspomagające bezpieczeństwo danych

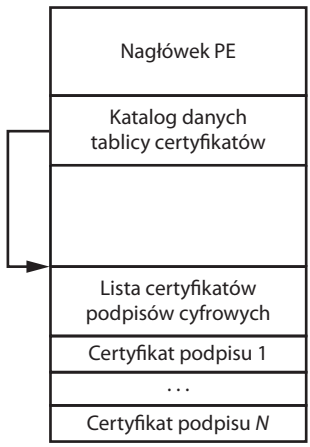
Zarejestruj się

Konferencję poprowadzi dla Was
dr Maciej Kawecki

Uwierzytelnianie plików wykonywalnych za pomocą podpisów cyfrowych

Jako pierwszy krok w stronę zrozumienia Secure Boot przyjrzyjmy się temu, jak w istocie podpisywany jest plik wykonywalny UEFI – czyli gdzie znajduje się podpis cyfrowy w tym pliku i jakie rodzaje podpisów obsługuje Secure Boot.

Dla plików wykonywalnych UEFI, które są obrazami Portable Executable (PE), podpisy cyfrowe są zawarte w specjalnych strukturach danych nazywanych *certyfikatami podpisów*. Lokalizacja tych certyfikatów w pliku binarnym jest określona specjalnym polem w strukturze danych nagłówka PE – katalogu danych tablicy certyfikatów (*Certificate Table Data Directory*), pokazanej na rysunku 17.1. Warto wspomnieć, że jeden plik może zawierać wiele podpisów cyfrowych wygenerowanych przy użyciu różnych kluczy podpisujących wykorzystywanych do różnych celów. Przez sprawdzenie tego pola oprogramowanie wbudowane UEFI może zlokalizować informacje o podpisie używanym do uwierzytelnienia pliku wykonywalnego.



Rysunek 17.1. Lokalizacja podpisów cyfrowych w obrazach UEFI

Inne typy obrazów wykonywalnych UEFI, takie jak obrazy *Terse Executable (TE)*, nie mają wbudowanych podpisów cyfrowych ze względu na specyfikę ich formatu wykonywalnego. Format obrazu TE został wyprowadzony z formatu PE/COFF w dążeniu do zredukowaniu rozmiaru TE, tak aby zajmował mniej miejsca. Tym samym obrazy TE zawierają tylko te pola formatu PE, które są niezbędne do wykonania obrazu w środowisku inicjowania platformy, co oznacza, że nie zawierają takich pól jak Certificate Table Data Directory. W rezultacie oprogramowanie układowe UEFI nie może bezpośrednio uwierzytelnić takich obrazów przez weryfikację ich podpisów cyfrowych. Niemniej jednak Secure Boot udostępnia możliwości uwierzytelniania tych obrazów przez użycie skrótów kryptograficznych, czyli mechanizmu, który opiszemy bardziej szczegółowo w następnym podrozdziale.

Układ osadzonego certyfikatu podpisu zależy od jego typu. Nie będziemy tu wnikać w specyfikę układu, ale więcej informacji można znaleźć w podrozdziale „Lokalizacja podpisów sterowników” na stronie 84.

Każdy typ certyfikatu podpisu używany w Secure Boot zawiera jako minimum następujące elementy: informację o algorytmach kryptograficznych użytych do generowania i weryfikacji podpisu (np. identyfikatory kryptograficznych funkcji skrótu i algorytmu podpisu cyfrowego), skrót kryptograficzny rozważanego pliku wykonywalnego, faktyczny podpis cyfrowy oraz klucz publiczny służący do weryfikacji podpisu.

Informacje te są wystarczające, aby Secure Boot zweryfikował autentyczność obrazu wykonywalnego. W tym celu oprogramowanie układowe UEFI lokalizuje i odczytuje certyfikat cyfrowy z pliku wykonywalnego, oblicza skrót pliku zgodnie ze wskazanym algorytmem, po czym porównuje uzyskany skrót z podanym w certyfikacie podpisu. Jeśli są zgodne, oprogramowanie UEFI weryfikuje cyfrowy podpis skrótu, używając klucza dostarczonego w certyfikacie podpisu. Jeśli weryfikacja podpisu zakończy się sukcesem, oprogramowanie UEFI akceptuje podpis. W każdym innym przypadku (takim jak niezgodność skrótów lub niepowodzenie weryfikacji podpisu), oprogramowanie UEFI nie jest w stanie uwierzytelnić obrazu.

Jednak proste zweryfikowanie tego, że podpisy pasują, nie wystarcza, aby ustanowić zaufanie do pliku wykonywalnego UEFI. Oprogramowanie układowe UEFI musi się również upewnić, że plik wykonywalny został podpisany autoryzowanym kluczem. W przeciwnym razie nic nie powstrzymywałoby kogokolwiek przed wygenerowaniem własnego klucza podpisującego i podpisania nim złośliwego obrazu, aby przeszedł walidację wykonywaną przez Secure Boot.

Z tego powodu klucz publiczny używany do walidacji podpisu musi być dopasowany do zaufanego klucza prywatnego. Oprogramowanie układowe UEFI jawnie ufa tym kluczom prywatnym, zatem mogą one zostać użyte do ustanowienia zaufania do obrazu. Lista zaufanych kluczy publicznych przechowywana jest w bazie danych `db`, którą zajmiemy się w następnej kolejności.

Baza danych db

Baza danych `db` przechowuje listę zaufanych certyfikatów kluczy publicznych, autoryzowanych do uwierzytelniania podpisów. Ilekroć Secure Boot wykonuje weryfikację podpisu pliku wykonywalnego, sprawdza klucz publiczny tego podpisu względem listy kluczy w bazie danych `db`, aby ustalić, czy może ufać temu kluczowi. Tylko kod podpisany kluczami prywatnymi odpowiadającymi tym certyfikatom zostanie wykonany w platformie podczas procesu rozruchu.

Jako uzupełnienie listy zaufanych certyfikatów kluczy publicznych baza danych `db` zawiera skróty poszczególnych plików wykonywalnych,

które są dopuszczone do wykonywania w danej platformie, niezależnie od tego, czy są one podpisane, czy nie. Ten mechanizm można wykorzystać do uwierzytelniania plików TE, które nie zawierają osadzonych podpisów cyfrowych.

Zgodnie ze specyfikacją UEFI baza danych podpisów jest przechowywana w zmiennej w nieulotnej pamięci (NVRAM), która jest zachowywana między ponownymi uruchomieniami systemu. Implementacja zmiennych NVRAM jest zależna od platformy i różni wytwórcy sprzętu (OEM) mogą ją implementować na różne sposoby. Najczęściej zmienne przechowywane są w tym samym układzie flash SPI, który zawiera oprogramowanie układowe platformy, takiej jak BIOS. Jak zobaczymy w podrozdziale „Modyfikowanie zmiennych UEFI w celu obejścia testów zabezpieczeń" na stronie 376, prowadzi to do podatności, których można użyć w celu ominięcia Secure Boot.

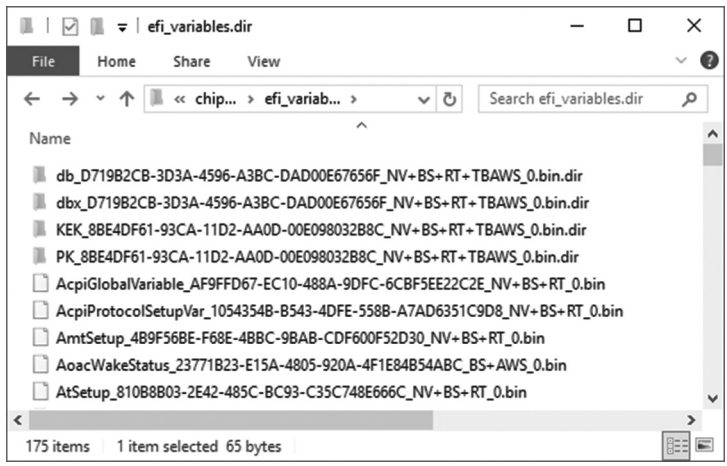
Zawartość bazy danych `db` we własnym systemie można zbadać, zrzucając zawartość zmiennej NVRAM, która ją zawiera. W naszym przykładzie użyjemy komputera Lenovo Thinkpad T540p, ale czytelnik może wykorzystać właśnie używany komputer. Zawartość zmiennej NVRAM zrzucimy przy użyciu zestawu narzędzi open source Chipsec, z którym spotkaliśmy się już w rozdziale 15. Zestaw ten zawiera bogatą funkcjonalność przydatną przy analizach śledczych i omówimy go bardziej szczegółowo w rozdziale 19.

Narzędzie Chipsec należy pobrać z repozytorium GitHub: *https://github.com/chipsec/chipsec/*. Narzędzie jest zależne od `winpy` (Python for Windows Extensions), które należy pobrać i zainstalować przed uruchomieniem Chipsec. Gdy mamy już oba składniki, otwieramy wiersz polecenia lub inny interpreter poleceń, po czym przechodzimy do katalogu zawierającego pobrane narzędzie Chipsec. Następnie, aby uzyskać listę zmiennych UEFI, należy wpisać następujące polecenie:

```
$ chipsec_util.py uefi var-list
```

Polecenie to zrzuca wszystkie zmienne UEFI z bieżącego katalogu do podkatalogu *efi_variables.dir* i dekoduje zawartość niektórych z nich (Chipsec dekoduje tylko zawartość znanych zmiennych). Po przejściu do tego katalogu powinniśmy zobaczyć coś podobnego do rysunku 17.2.

Każdy wpis w tym katalogu odpowiada oddzielnej zmiennej UEFI NVRAM. Zmienne te mają strukturę *`VarName_VarGUID_VarAttributes.bin`*, gdzie *`VarName`* to nazwa zmiennej, *`VarGUID`* to 16 -bajtowy globalnie unikatowy identyfikator (GUID) tej zmiennej, a *`VarAttributes`* jest listą atrybutów zmiennej w postaci skróconej. Opierając się na specyfikacji UEFI, możemy wyjaśnić niektóre atrybuty wpisów widocznych na rysunku 17.2.



Rysunek 17.2. Zmienne UEFI zrzucone przez Chipsec

`NV` – nieulotna (*nonvolatile*), co oznacza, że zawartość zmiennej jest zachowywana w czasie ponownego uruchamiania;

`BS` – może być dostępna dla usług rozruchowych UEFI; usługi rozruchowe UEFI są w ogólności dostępne w czasie rozruchu, przed uruchomieniem loadera systemu operacyjnego; po wywołaniu loadera usługi rozruchu UEFI nie są już dostępne;

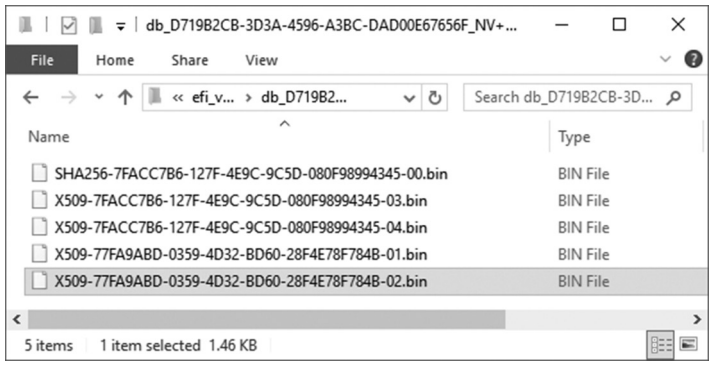
`RT` – może być dostępny dla usług UEFI czasu działania; w odróżnieniu od usług rozruchowych UEFI usługi czasu działania pozostają dostępne w czasie ładowania systemu operacyjnego i w trakcie jego działania;

`AWS` – zmienna uwierzytelniona licznikowo, co oznacza, że dowolna nowa zawartość tej zmiennej musi być podpisana autoryzowanym kluczem, aby zmienna mogła być zapisana; podpisane dane zmienne zawierają licznik, aby przeciwdziałać atakom tzw. atakom wstecznym (*rollback attacks*);

`TBAWS` – zmienna uwierzytelniona czasowo, co oznacza, że każda nowa zawartość zmiennej musi być podpisana autoryzowanym kluczem, aby zmienna mogła być zapisana; sygnatura czasowa podpisu odzwierciedla czas, w którym dane zostały podpisane; używana jest do potwierdzenia, że podpis został utworzony przed wygaśnięciem odpowiadającego mu klucza podpisującego (więcej informacji na temat uwierzytelniania opartego na czasie zawiera kolejny podrozdział).

Jeśli Secure Boot jest skonfigurowany i w danej platformie istnieje zmienna `db`, powinniśmy znaleźć w tym katalogu podfolder o nazwie rozpoczynającej się od *db_D719B2CB-3D3A-4596-A3BC-DA D00E67656F*. Gdy Chipsec zrzuca zmienną UEFI `db`, automatycznie dekoduje jej zawartość do tego podfolderu, zawierającego pliki odpowiadające certyfikatom kluczy

publicznych oraz skrótom autoryzowanym do wykonania obrazów UEFI. W naszym przypadku mamy tu pięć plików – cztery certyfikaty oraz jeden skrót SHA256, pokazane na rysunku 17.3.



Rysunek 17.3. Zawartość zmiennej bazy danych podpisów UEFI

Certyfikaty te są zakodowane przy użyciu X.509, standardu kryptograficznego definiującego format certyfikatów kluczy publicznych. Możemy je zdekodować, aby uzyskać informację o wystawcy, a zatem dowiedzieć się, czyj podpis pozwala przejść weryfikację Secure Boot. W tym celu użyliśmy przybornika `openssl`, opisanego w ramce „Przybornik OpenSSL". Narzędzie należy zainstalować ze strony *https://github.com/openssl/openssl/*, a następnie uruchomić następującym poleceniem, zastępując *`certificate_file_path`* katalogiem zawierającym certyfikat:

```
$ openssl x509 -in certificate_file_path
```

W systemie operacyjnym Windows można po prostu zmienić rozszerzenie nazwy pliku certyfikatu X.509 z *bin* na *crt*, po czym otworzyć plik przy użyciu Eksploratora, aby zobaczyć rezultat dekodowania. Tabela 17.1 zawiera uzyskane przez nas wyniki z wystawcami i przeznaczeniem certyfikatów.

Tabela 17.1. Zdekodowane certyfikaty i skróty ze zmiennej UEFI

Nazwa pliku	Wystawiony dla	Wystawiony przez
X509-7FACC7B6 -127F-4E9C-9C5D-080F98994345-03.bin	Thinkpad Product CA 2012	Lenovo Ltd. Root CA 2012
X509-7FACC7B6 -127F-4E9C-9C5D-080F98994345-04.bin	Lenovo UEFI CA 2014	Lenovo UEFI CA 2014
X509-77FA9ABD -0359-4D32-BD60 -28F4E78F784B -01.bin	Microsoft Corporation UEFI CA 2011	Microsoft Corporation Third-Party Marketplace Root
X509-77FA9ABD -0359-4D32-BD60 -28F4E78F784B -02.bin	Microsoft Windows Production PCA 2011	Microsoft Root Certificate Authority 2010

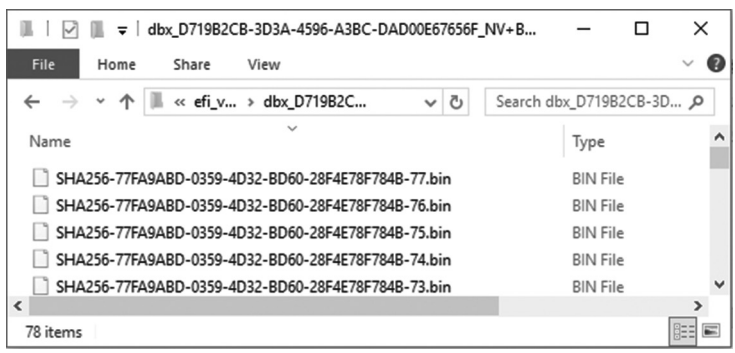
W tabeli tej można zobaczyć, że tylko obrazy UEFI podpisane przez Lenovo i Microsoft przejdą testy integralności kodu wykonywane przez UEFI Secure Boot.

> **Przybornik OpenSSL**
>
> OpenSSL jest biblioteką open source, która implementuje protokoły Secure Socket Layer oraz Transport Layer Security, a także algorytmy kryptograficzne ogólnego stosowania. Jest rozpowszechniana zgodnie z licencją w stylu Apache i często używana w komercyjnych oraz niekomercyjnych zastosowaniach. Biblioteka oferuje bogatą funkcjonalność wykorzystywaną do pracy z certyfikatami X.509, zarówno do analizy składni istniejących certyfikatów, jak i generowania nowych. Informacje na temat tego projektu są dostępne na stronie https://www.openssl.org/.

Baza danych dbx

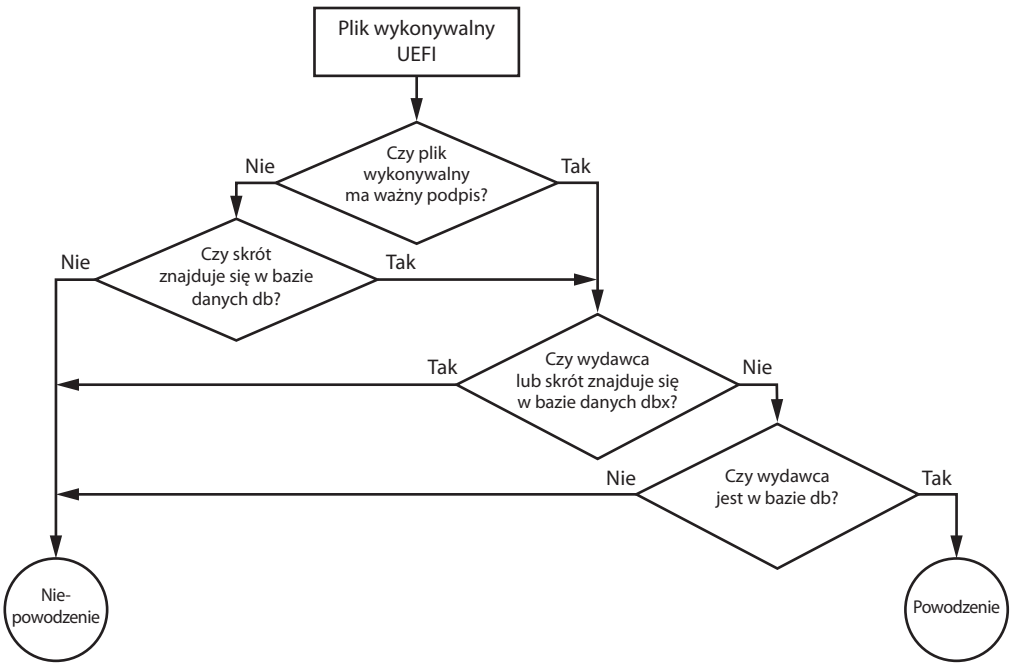
W przeciwieństwie do `db` baza danych `dbx` zawiera certyfikaty kluczy publicznych i skróty plików wykonywalnych UEFI, których uruchamianie w trakcie rozruchu jest zakazane. Baza ta jest również nazywana bazą danych odwołanych podpisów (*Revoked Signature Database*) i jawnie wylicza obrazy, które nie przejdą weryfikacji Secure Boot, powstrzymując wykonanie modułu o znanych podatnościach, które mogłyby przełamać zabezpieczenia całej platformy.

Zawartość bazy danych `dbx` można zbadać tak samo jak bazy podpisów `db`. Wśród folderów wygenerowanych po uruchomieniu narzędzia Chipsec znajdziemy folder *efi_variables.dir*, który powinien zawierać podfolder o nazwie rozpoczynającej się od *dbx_D719B2CB-3D3A-4596-A3BCDAD00E67656f*. Folder ten zawiera certyfikaty i skróty zabronionych obrazów UEFI. W naszym przypadku folder zawierał jedynie 78 skrótów, ale żadnych certyfikatów, co widać na rysunku 17.4.



Rysunek 17.4. Zawartość bazy danych dbx (bazy danych odwołanych podpisów) zmiennej UEFI

Na rysunku 17.5 jest pokazany algorytm weryfikacji podpisu obrazu, wykorzystujący obie bazy danych `db` i `dbx`.



Rysunek 17.5. Algorytm weryfikacji obrazu UEFI wykonywanej przez Secure Boot

Na tym rysunku widzimy, że plik wykonywalny UEFI przechodzi uwierzytelnienie tylko wtedy, gdy jego skrót lub certyfikat podpisu jest zaufany na podstawie bazy danych `db` i że nie jest on wymieniony w bazie danych `dbx`. W przeciwnym wypadku obraz nie przejdzie testu integralności Secure Boot.

Uwierzytelnienie oparte na czasie

Oprócz baz danych `db` i `dbx` Secure Boot używa dwóch innych baz danych o nazwach `dbr` i `dbt`. Pierwsza z nich, `dbr`, zawiera certyfikaty klucza publicznego służące do weryfikowania sygnatur programu ładującego przywracania systemu operacyjnego. Nie będziemy się nią tu zajmować.

Druga, `dbt`, zawiera certyfikaty sygnatur czasowych służące do walidacji sygnatury czasowej podpisu cyfrowego pliku wykonywalnego UEFI, umożliwiając wykonanie uwierzytelniania opartego na czasie (TBAWS) w Secure Boot. (O TBAWS wspomnieliśmy wcześniej w tym rozdziale przy przeglądaniu atrybutów zmiennych UEFI).

Cyfrowy podpis pliku wykonywalnego UEFI niekiedy zawiera sygnaturę czasową nadaną przez usługę *Time Stamping Authority (TSA)*. Sygnatura czasowa

podpisu odzwierciedla czas, w którym podpis został wygenerowany. Porównując sygnaturę czasową podpisu i sygnaturę czasową klucza podpisującego, Secure Boot ustala, czy podpis został wygenerowany przed, czy po wygaśnięciu klucza podpisującego. W ogólności, data wygaśnięcia (*expiration*) klucza podpisującego jest datą, po której klucz podpisujący jest uznawany za nieważny. W rezultacie sygnatura czasowa podpisu pozwala mechanizmowi Secure Boot zweryfikować, czy podpis został wygenerowany w chwili, gdy klucz podpisujący nie został unieważniony, co gwarantuje, że podpis jest uprawniony. W ten sposób uwierzytelnianie oparte na czasie redukuje złożoność PKI, jeśli chodzi o certyfikaty zawarte w bazie danych `db` Secure Boot.

Uwierzytelnianie oparte na czasie pozwala również uniknąć ponownego podpisywania tych samych obrazów UEFI. Sygnatura czasowa podpisu stanowi dowód dla Secure Boot, że obraz UEFI został podpisany przed wygaśnięciem lub odwołaniem odpowiadającego mu klucza podpisującego. W rezultacie podpis pozostaje ważny nawet po wygaśnięciu klucza podpisującego, gdyż został utworzony w czasie, gdy klucz ten był nadal ważny i nie został złamany.

Klucze Secure Boot

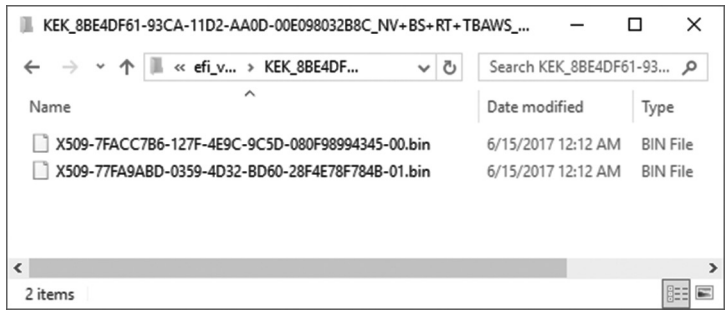
Teraz, gdy już wiemy, skąd Secure Boot czerpie informacje o zaufanych i odwołanych certyfikatach kluczy publicznych, zajmijmy się tym, jak te bazy danych są przechowywane i chronione przed nieautoryzowaną modyfikacją. W końcu przez modyfikację bazy danych `db` napastnik mógłby łatwo ominąć testy Secure Boot, wstrzykując złośliwy certyfikat i zastępując program ładujący systemu operacyjnego podstępnym programem podpisanym kluczem prywatnym odpowiadającym temu certyfikatowi. A skoro złośliwy certyfikat znajduje się w bazie podpisów `db`, Secure Boot pozwoliłby na uruchomienie podstępnego programu ładującego.

Tak więc w celu ochrony baz danych `db` i `dbx` przed nieautoryzowaną modyfikacją dostawca platformy lub systemu operacyjnego musi je podpisać. Gdy oprogramowanie układowe UEFI przechodzi do odczytania zawartości tych baz danych, najpierw je uwierzytelnia, weryfikując ich podpis cyfrowy przy użyciu klucza publicznego nazywanego *kluczem wymiany kluczy* (*key exchange key* – KEK). Następnie uwierzytelnia każdy KEK drugim kluczem, nazywanym *kluczem platformy* (*platform key* – PK).

Klucze wymiany kluczy

Lista publicznych kluczy wymiany kluczy, podobnie jak bazy danych `db` i `dbx`, jest przechowywana w zmiennej NVRAM UEFI. Zbadamy zawartość zmiennej KEK, używając wyników poprzedniego wykonania polecenia `chipsec`. Otwieramy katalog zawierający wyniki i powinniśmy zobaczyć w nim podfolder opisany jako coś podobnego do *KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C*, zawierający certyfikaty publicznych KEK (rysunek 17.6).

Ta zmienna UEFI również jest uwierzytelniana, co zobaczymy za chwilę.



Rysunek 17.6. Zawartość zmiennej KEK

Jedynie właściciel klucza prywatnego odpowiadającego dowolnemu z tych certyfikatów może modyfikować zawartość baz danych `db` i `dbx`. W tym przykładzie mamy tylko dwa certyfikaty KEK, wystawione przez Microsoft i Lenovo, co widać w tabeli 17.2.

Tabela 17.2. Certyfikaty w zmiennej KEK UEFI

Nazwa pliku	Wystawiony dla	Wystawiony przez
X509-7FACC7B6-127F-4E9C-9C5D -080F98994345-00.bin	Lenovo Ltd. KEK CA 2012	Lenovo Ltd. KEK CA 2012
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	Microsoft Corporation KEK CA 2011	Microsoft Corporation Third-Party Marketplace Root

Można znaleźć właścicieli kluczy prywatnych odpowiadających certyfikatom KEK w systemie, zrzucając zmienną KEK i wykonując użyte wcześniej polecenie `openssl`.

Klucz platformy

Klucz platformy jest ostatnim kluczem podpisującym w hierarchii kluczy PKI mechanizmu Secure Boot. Jak można się domyślić, klucz ten służy do uwierzytelnienia klucza wymiany kluczy przez podpisanie zmiennej KEK UEFI. Zgodnie ze specyfikacją UEFI każda platforma ma jeden klucz platformy. Zazwyczaj klucz ten odpowiada producentowi platformy.

Wróćmy do podfolderu *PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C* w katalogu *efi_variables.dir*, który został utworzony po wykonaniu polecenia `chipsec`. Możemy w nim znaleźć certyfikat publicznego klucza platformy. Certyfikat ten odpowiada używanej platformie. Dlatego, ponieważ użyliśmy komputera Lenovo Thinkpad T540p, możemy oczekiwać, że nasz certyfikat klucza platformy będzie odpowiadać firmie Lenovo (rysunek 17.7).

Rysunek 17.7. Certyfikat PK

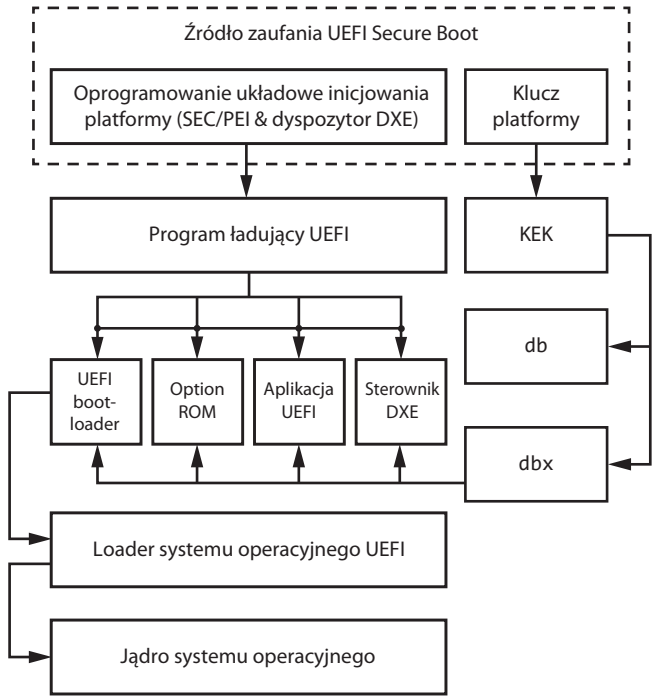
Jak widać, że nasz został istotnie wystawiony przez Lenovo. Zmienna PK UEFI również jest uwierzytelniona i każda aktualizacja tej zmiennej powinna być podpisana odpowiednim kluczem prywatnym. Innymi słowy, jeśli właściciel platformy (albo producent platformy w terminologii UEFI) chce uaktualnić zmienną PK przy użyciu nowego certyfikatu, bufor z nowym certyfikatem powinien zostać podpisany kluczem prywatnym, który odpowiada bieżącemu certyfikatowi przechowywanemu w zmiennej PK.

UEFI Secure Boot: pełny obraz

Teraz, gdy poznaliśmy już pełną hierarchię infrastruktury PKI używanej w UEFI Secure Boot, spróbujmy to zebrać razem, aby zobaczyć pełny obraz, pokazany na rysunku 17.8.

U góry rysunku możemy zobaczyć, że źródło zaufania (komponenty, którym UEFI Secure Boot ufa i na których opierają się wszystkie późniejsze weryfikacje) to oprogramowanie układowe inicjowania platformy oraz klucz platformy. Oprogramowanie układowe inicjowania platformy to pierwszy fragment kodu wykonywanego po wyjściu procesora ze stanu resetu i UEFI Secure Boot domyślnie ufa temu kodowi. Jeśli napastnik zdoła przełamać oprogramowanie układowe inicjowania platformy, to złamany zostanie cały łańcuch zaufania wymuszany przez Secure Boot. W takim przypadku na-

pastnik może zmodyfikować dowolny moduł UEFI, który implementuje procedury Secure Boot weryfikacji obrazów, tak by zawsze zwracał sukces, a w rezultacie pozwalał na przejście uwierzytelniania przez każdy dostarczony obraz UEFI.



Rysunek 17.8. Przepływ weryfikacji w UEFI Secure Boot

U góry rysunku możemy zobaczyć, że źródło zaufania (komponenty, którym UEFI Secure Boot ufa i na których opierają się wszystkie późniejsze weryfikacje) to oprogramowanie układowe inicjowania platformy oraz klucz platformy. Oprogramowanie układowe inicjowania platformy to pierwszy fragment kodu wykonywanego po wyjściu procesora ze stanu resetu i UEFI Secure Boot domyślnie ufa temu kodowi. Jeśli napastnik zdoła przełamać oprogramowanie układowe inicjowania platformy, to złamany zostanie cały łańcuch zaufania wymuszany przez Secure Boot. W takim przypadku napastnik może zmodyfikować dowolny moduł UEFI, który implementuje procedury Secure Boot weryfikacji obrazów, tak by zawsze zwracał sukces, a w rezultacie pozwalał na przejście uwierzytelniania przez każdy dostarczony obraz UEFI.

To dlatego model zaufania Secure Boot zakłada, że poprawnie zaimplementowaliśmy mechanizm Firmware Secure Update (zabezpieczonej aktualizacji oprogramowania układowego), który wymagał, aby każda

aktualizacja oprogramowania układowego była podpisana właściwym kluczem podpisującym (który musi się różnić od klucza platformy). W ten sposób mogą następować tylko autoryzowane aktualizacje oprogramowania inicjowania platformy i źródło zaufania pozostaje nienaruszone.

Można łatwo zauważyć, że ten model zaufania nie chroni przed napastnikiem dysponującym fizycznym dostępem, który może fizycznie przeprogramować układ flash SPI złośliwym obrazem oprogramowania układowego i przejąć oprogramowanie inicjowania platformy. Zagadnieniem ochrony oprogramowania układowego przed atakami fizycznymi zajmiemy się w podrozdziale „Ochrona Secure Boot za pomocą weryfikowanego lub mierzonego rozruchu" na stronie 377.

W górnej części rysunku 17.8 możemy zauważyć, że klucz platformy dostarczany przez jej producenta ma ten sam poziom naturalnego zaufania, co oprogramowanie układowe inicjowania platformy. Klucz ten jest używany do ustanowienia zaufania między oprogramowaniem układowym inicjowania platformy a producentem platformy. Gdy zostanie dostarczony klucz platformy, oprogramowanie układowe pozwoli producentowi na aktualizację klucza wymiany kluczy, a w rezultacie kontrolowanie, które obrazy mają przejść testy Secure Boot, a które nie.

Jeden poziom niżej widzimy klucze wymiany kluczy, które ustanawiają zaufanie między oprogramowaniem układowym inicjowania platformy a systemem operacyjnym uruchomionym na platformie. Gdy tylko klucze wymiany kluczy platformy są umieszczone w zmiennej UEFI, system operacyjny jest w stanie określić, które obrazy mogą przejść test Secure Boot. Na przykład, dostawca systemu operacyjnego może użyć klucza wymiany kluczy, aby zezwolić oprogramowaniu układowemu UEFI na wykonanie programu ładującego system.

Na dole modelu zaufania widzimy bazy danych `db` i `dbx` podpisane przy użyciu kluczy wymiany kluczy, które zawierają skróty obrazów i certyfikaty kluczy publicznych wykorzystywane bezpośrednio w testach integralności plików wykonywalnych, wymuszanych przez Secure Boot.

Zasady Secure Boot

Jak widzimy, sam z siebie mechanizm Secure Boot używa zmiennych `PK`, `KEK`, `db`, `dbx` oraz `dbt`, aby poinformować platformę, czy dany obraz wykonywalny jest zaufany. Jednak to, jak zostanie zinterpretowany wynik weryfikacji wykonanej przez Secure Boot (innymi słowy to, czy dany obraz zostanie wykonany, czy nie), zależy głównie od stosowanej zasady.

Kilkukrotnie wspominaliśmy już w tym rozdziale o zasadach Secure Boot bez wchodzenia w szczegóły, czym one w istocie są. Przyjrzyjmy się bliżej tej koncepcji.

Mówiąc zasadniczo, zasada Secure Boot nakazuje, jakie działania powinno podjąć oprogramowanie układowe platformy po wykonaniu uwierzytelnienia obrazu. Oprogramowanie układowe może wykonać obraz, odmówić jego wykonania, odłożyć wykonanie albo poprosić użytkownika o podjęcie decyzji.

Zasada Secure Boot nie jest ściśle zdefiniowana w specyfikacji UEFI, a tym samym jest zależna od implementacji. W szczególności zasady mogą różnić się między implementacjami oprogramowania układowego UEFI różnych dostawców. W tym podrozdziale zbadamy niektóre elementy zasady Secure Boot implementowane w kodzie źródłowym EDK 2 firmy Intel, którego używaliśmy w rozdziale 15. Warto teraz pobrać lub sklonować kod źródłowy EDK 2 z repozytorium dostępnego pod adresem *https://github.com/tianocore/edk2/*, jeśli czytelnik jeszcze tego nie zrobił.

Jednym z elementów, które bierze pod uwagę Secure Boot w wersji implementowanej w EDK 2, jest pochodzenie obrazów wykonywalnych podlegających uwierzytelnieniu. Obrazy te mogą pochodzić z różnych urządzeń pamięci, z których niektóre mogą być z natury zaufane. Na przykład, jeśli obraz jest ładowany z układu flash SPI, co oznacza, że jest zlokalizowany w tym samym urządzeniu pamięci, co reszta oprogramowania układowego UEFI, platforma może mu ufać automatycznie. (Niemniej jednak, jeśli napastnik jest w stanie zmienić obraz w układzie SPI, może również zmodyfikować resztę oprogramowania układowego i zupełnie wyłączyć Secure Boot. Atak taki omówimy dalej w podrozdziale „Łatanie oprogramowania inicjowania platformy w celu wyłączenia Secure Boot" na stronie 374). Z drugiej strony, jeśli obraz jest ładowany z zewnętrznego urządzenia PCI – jak na przykład Option ROM, czyli specjalne oprogramowanie układowe ładowane z zewnętrznych urządzeń peryferyjnych w środowisku przeduruchomieniowym – będzie traktowane jako niezaufane i podlegające sprawdzeniu przez Secure Boot.

Przedstawimy teraz w zarysie definicje niektórych zasad, które określają, jak przetwarzane są obrazy w zależności od ich pochodzenia. Zasady te można znaleźć w pliku *SecurityPkg\SecurityPkg.dec* w repozytorium EDK 2. Każda zasada przypisuje domyślną wartość obrazom, które spełniają dane kryterium.

`PcdOptionRomImageVerificationPolicy` – definiuje zasadę weryfikacji dla obrazów ładowanych jako Option ROM, takich jak pochodzące z urządzeń PCI (domyślna wartość: 0x00000004);

`PcdRemovableMediaImageVerificationPolicy` – definiuje zasadę weryfikacji dla obrazów zlokalizowanych na nośnikach wymiennych, takich jak CD-ROM, USB oraz sieć (domyślna wartość: 0x00000004);

`PcdFixedMediaImageVerificationPolicy` – definiuje zasadę weryfikacji dla obrazów zlokalizowanych na urządzeniach ze stałymi nośnikami, takich jak dyski twarde (domyślna wartość: 0x00000004).

Oprócz tych zasad istnieją jeszcze dwie dodatkowe zasady, które nie są jawnie zdefiniowane w pliku *SecurityPkg\SecurityPkg.dec*, ale są używane w implementacji Secure Boot w EDK 2:

flash SPI ROM – definiuje zasadę weryfikacji dla obrazów zlokalizowanych w układzie flash SPI (domyślna wartość: 0x00000000);

inne pochodzenie – definiuje zasadę weryfikacji dla dowolnych obrazów zlokalizowanych na urządzeniach innych niż właśnie opisane (domyślna wartość: 0x00000004).

UWAGA *Trzeba pamiętać, że nie jest to wyczerpująca lista zasad Secure Boot używanych przy uwierzytelnianiu obrazów. Różni dostawcy oprogramowania układowego mogą modyfikować lub rozszerzać tę listę o własne zasady.*

Oto opisy domyślnych wartości zasady:

0x00000000 – zawsze ufaj obrazowi, niezależnie od tego, czy jest on podpisany i czy jego skrót znajduje się w bazie danych `db` lub `dbx`;

0x00000001 – nigdy nie ufaj obrazowi; nawet obrazy z poprawnymi podpisami zostaną odrzucone;

0x00000002 – zezwalaj na wykonanie, gdy nastąpi naruszenie zabezpieczeń; obraz zostanie wykonany nawet wtedy, gdy podpis nie może być zweryfikowany lub jego skrót jest wymieniony na czarne liście w bazie danych `dbx`;

0x00000003 – odłóż wykonanie, gdy nastąpi naruszenie zabezpieczeń; w tym przypadku obraz nie jest odrzucany natychmiast i zostaje załadowany do pamięci, niemniej jednak jego wykonanie jest opóźnione do chwili ponownego sprawdzenia jego statusu uwierzytelnienia;

0x00000004 – odmawiaj wykonania, gdy Secure Boot nie zdoła uwierzytelnić obrazu przy użyciu baz danych `db` i `dbx`;

0x00000005 – zapytaj użytkownika, gdy wystąpi naruszenie zabezpieczeń; w tym przypadku, jeśli Secure Boot nie zdoła uwierzytelnić obrazu, autoryzowany użytkownik może podjąć decyzję, czy należy ufać temu obrazowi – użytkownik może, na przykład, zobaczyć komunikat z pytaniem podczas rozruchu.

Na podstawie definicji zasad Secure Boot możemy zobaczyć, że wszystkie obrazy ładowane z układu flash SPI są naturalnie zaufane i w ogóle nie podlegają weryfikacji podpisów cyfrowych. We wszystkich innych przypadkach domyślna wartość 0x000000004 wymusza weryfikację podpisów i zabrania wykonania dowolnego nieuwierzytelnionego kodu, który został dostarczony jako Option ROM lub jest zlokalizowany na nośniku wymiennym, stałym lub dowolnym innym.

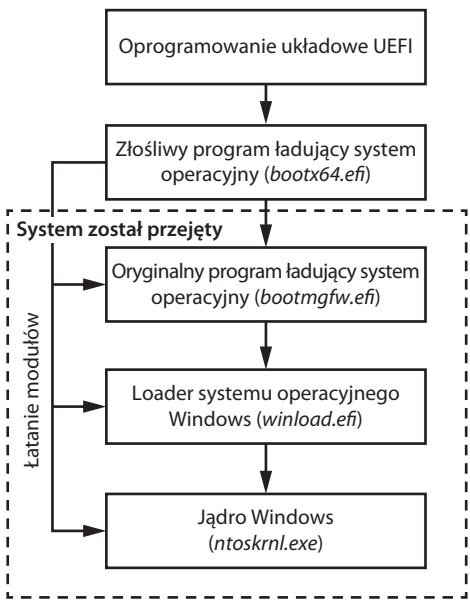
Ochrona przed bootkitami przy użyciu Secure Boot

Teraz, gdy wiemy jak działa Secure Boot, przyjrzyjmy się konkretnemu przykładowi, jak chroni on przed bootkitami biorącymi za cel przebieg rozruchu systemu operacyjnego. Nie będziemy omawiać bootkitów, które atakują MBR i VBR, gdyż, jak wyjaśniliśmy w rozdziale 14, oprogramowanie układowe UEFI nie używa już takich obiektów jak MBR i VBR (z wyjątkiem działania w trybie kompatybilności UEFI), zatem tradycyjne bootkity nie mogą przełamać systemów opartych na UEFI.

Jak wspomnieliśmy w rozdziale 15, bootkit DreamBoot był pierwszym publicznym dowodem koncepcji bootkitu atakującego systemy oparte na UEFI. W systemie UEFI bez obecnego mechanizmu Secure bootkit ten działa następująco:

(1) Autor bootkitu zastępuje oryginalny program ładujący UEFI dla systemu Windows, *bootmgfw.efi*, złośliwym programem ładującym *bootx64.efi*, na partycji rozruchowej.

(2) Złośliwy program ładujący wczytuje oryginalny *bootmgfw.efi*, łata go, aby uzyskać kontrolę nad loaderem Windows *winload.efi*, po czym go wykonuje, co jest pokazane na rysunku 17.9.



Rysunek 17.9. Przebieg ataku DreamBoot na program ładujący systemu operacyjnego

(3) Złośliwy kod kontynuuje łatanie modułów systemowych, aż osiągnie jądro systemu operacyjnego, omijając mechanizmy ochrony jądra (takie jak zasada podpisywania kodu trybu jądra) mające na celu uniemożliwienie wykonania nieautoryzowanego kodu trybu jądra.

Ten typ ataku jest możliwy, gdyż domyślnie program ładujący system operacyjny nie jest uwierzytelniany w procesie rozruchu UEFI. Oprogramowanie układowe UEFI uzyskuje ze zmiennej UEFI lokalizację programu ładującego, którą w przypadku platform Microsoft Windows jest *\EFI\Microsoft\Boot\bootmgfw.efi* na partycji rozruchowej. Napastnik z uprawnieniami systemowymi może łatwo zastąpić lub zmienić program ładujący.

Jednakże gdy Secure Boot jest włączony, atak ten nie jest już możliwy. Ponieważ Secure Boot weryfikuje integralność obrazów UEFI wykonywanych w czasie rozruchu, a program ładujący system operacyjny jest jednym z weryfikowanych plików wykonywalnych, Secure Boot sprawdzi podpis programu ładującego względem baz danych `db` i `dbx`. Złośliwy program ładujący nie jest podpisany właściwym kluczem podpisującym, zatem potencjalnie nie przejdzie testu i nie zostanie uruchomiony (zależnie od zasady rozruchowej). To jeden ze sposobów, jakim Secure Boot chroni przed bootkitami.

Atakowanie Secure Boot

Przyjrzyjmy się teraz pewnym atakom przeciwko UEFI Secure Boot, które mogą się powieść. Ponieważ Secure Boot polega na oprogramowaniu układowym inicjowania platformy i kluczu platformy jako źródłach zaufania, jeśli któryś z tych komponentów zostanie złamany, cały łańcuch sprawdzeń wykonywanych przez Secure Boot staje się bezużyteczny. Przyjrzymy się zarówno bootkitom, jak i rootkitom zdolnym do zneutralizowania Secure Boot.

Bootkity, którymi zajmiemy się tutaj, głównie opierają się na modyfikacjach zawartości pamięci flash SPI. W nowoczesnych systemach komputerowych układy flash SPI często służą jako główne miejsce przechowywania oprogramowania układowego. Niemal każdy laptop i komputer biurkowy przechowuje oprogramowanie UEFI w pamięci flash, do której dostęp następuje za pośrednictwem kontrolera SPI.

W rozdziale 15 przedstawiliśmy różne ataki, które instalują trwałe rootkity UEFI w oprogramowaniu układowym w pamięci flash, zatem nie będziemy ponownie omawiać tych szczegółów, choć te same ataki (problemy procedury obsługi SMI, skryptu rozruchowego S3, ochrony zapisu BIOS-u itd.) mogą zostać użyte również przeciwko Secure Boot. W przypadku ataków omawianych w tym podrozdziale zakładamy, że napastnik jest już w stanie modyfikować zawartość pamięci flash zawierającej oprogramowanie UEFI. Przyjrzyjmy się, co mógłby zrobić potem!

Łatanie oprogramowania PI w celu wyłączenia Secure Boot

Gdy napastnik jest w stanie zmodyfikować zawartość pamięci flash SPI, może łatwo wyłączyć Secure Boot, łatając oprogramowanie inicjowania platformy.

Na rysunku 17.8 widzieliśmy, że cały mechanizm UEFI Secure Boot jest zamocowany w oprogramowaniu inicjowania platformy, zatem jeśli zmienimy moduły oprogramowania inicjowania platformy implementujące Secure Boot, faktycznie możemy wyłączyć jego funkcjonalność.

Aby zbadać ten proces, jako przykładowej implementacji UEFI ponownie użyjemy kodu źródłowego EDK 2 Intela (*https://github.com/tianocore/edk2/*). Możemy tam sprawdzić, gdzie implementowana jest funkcjonalność weryfikacji Secure Boot i jak moglibyśmy ją popsuć.

W folderze *SecurityPkg/Library/DxeImageVerificationLib* w repozytorium znajdziemy plik kodu źródłowego *DxeImageVerificationLib.c*, który implementuje funkcjonalność weryfikacji integralności kodu. Mówiąc ściślej, plik zawiera procedurę `DxeImageVerificationHandler`, która decyduje, czy plik wykonywalny UEFI jest zaufany i powinien być wykonany, czy też nie przeszedł weryfikacji. Listing 17.1 zawiera prototyp tej procedury.

```
EFI_STATUS EFI_API DxeImageVerificationHandler (
  IN  UINT32                                AuthenticationStatus, ❶
  IN  CONST EFI_DEVICE_PATH_PROTOCOL        *File, ❷
  IN  VOID                                  *FileBuffer, ❸
  IN  UINTN                                 FileSize, ❹
  IN  BOOLEAN                               BootPolicy ❺
);
```

Listing 17.1. Definicja procedury `DxeImageVerificationHandler`

Jako pierwszy parametr procedura przyjmuje zmienną `AuthenticationStatus` ❶, która wskazuje, czy obraz jest podpisany. Argument `File` ❷ jest wskaźnikiem do ścieżki urządzenia wysyłanego pliku. Argumenty `FileBuffer` ❸ oraz `FileSize` ❹ zawierają wskaźnik do obrazu UEFI oraz jego rozmiar używane do weryfikacji.

Na koniec `BootPolicy` ❺ jest parametrem wskazującym, czy żądanie załadowania uwierzytelnianego właśnie obrazu pochodzi od menedżera rozruchu UEFI i jest wyborem rozruchu (co oznacza, że obraz ten jest programem ładującym wybrany system operacyjny). Bardziej szczegółowe omówienie menedżera rozruchu UEFI zamieściliśmy w rozdziale 14.

Po zakończeniu weryfikacji procedura zwraca jedną z następujących wartości:

`EFI_SUCCESS` – uwierzytelnienie przebiegło z powodzeniem i obraz zostanie wykonany.

`EFI_ACCESS_DENIED` – obraz nie został uwierzytelniony, gdyż zasada platformy określa, że oprogramowanie układowe nie może użyć tego pliku obrazu. Może to nastąpić, gdy oprogramowanie układowe próbuje załadować obraz z nośnika wymiennego, a zasada platformy zabrania

wykonywania kodu z nośników wymiennych w czasie rozruchu bez względu na to, czy są one podpisane, czy nie. W takim przypadku procedura natychmiast zwraca `EFI_ACCESS_DENIED` bez żadnego weryfikowania podpisów.

`EFI_SECURITY_VIOLATION` – uwierzytelnienie się nie powiodło albo dlatego, że Secure Boot nie był w stanie zweryfikować podpisu cyfrowego, albo dlatego, że wartość skrótu pliku wykonywalnego znajduje się w bazie danych nieuprawnionych obrazów (`dbx`). Ta wartość wynikowa sygnalizuje, że obraz nie jest zaufany i platforma powinna postępować zgodnie z zasadą Secure Boot, aby ustalić, czy obraz ma zostać uruchomiony.

`EFI_OUT_RESOURCE` – w trakcie procesu weryfikacji wystąpił błąd z powodu braku zasobów systemowych (zazwyczaj niedostatecznej pamięci) do wykonania uwierzytelnienia obrazu.

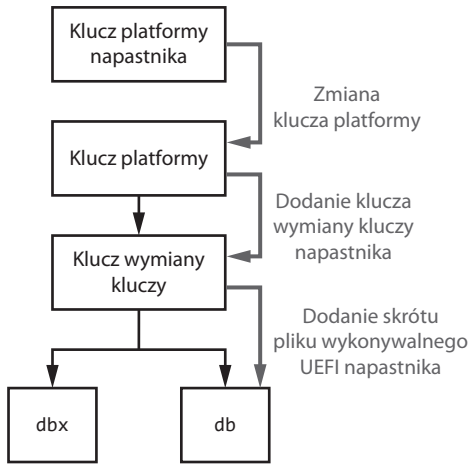
Aby ominąć testy Secure Boot, napastnik z dostępem do zapisu w pamięci flash SPI może załatać tę procedurę, aby zawsze zwracała wartość `EFI_SUCCESS` dla dowolnego pliku wykonywalnego, który otrzyma jako wejście. W rezultacie wszystkie obrazy UEFI będą przechodziły uwierzytelnianie, niezależnie od tego, czy są podpisane, czy nie.

Modyfikowanie zmiennych UEFI w celu obejścia testów zabezpieczeń

Innym sposobem zaatakowania implementacji Secure Boot jest modyfikacja zmiennych NVRAM. Jak wyjaśniliśmy wcześniej w tym rozdziale, Secure Boot używa określonych zmiennych do przechowania swoich parametrów konfiguracyjnych, czyli takich szczegółów jak to, czy Secure Boot jest włączony, klucz platformy i klucz wymiany kluczy, bazy danych podpisów oraz zasady platformy. Jeśli napastnik może zmodyfikować te zmienne, może wyłączyć albo pominąć testy weryfikacyjne Secure Boot.

W istocie większość implementacji Secure Boot przechowuje zmienne UEFI NVRAM w pamięci flash SPI wraz z systemowym oprogramowaniem układowym. Mimo że zmienne te są uwierzytelniane i zmiana ich wartości z poziomu trybu jądra przy użyciu API UEFI wymaga dysponowania odpowiednim kluczem prywatnym, napastnik dysponujący możliwością zapisu do pamięci flash SPI może zmienić ich zawartość.

Gdy napastnik ma już dostęp do zmiennych UEFI NVRAM, może na przykład majstrować przy `PK`, `KEK`, `db` i `dbx`, aby dodać własne, podstępne certyfikaty, które pozwolą złośliwemu modułowi pominąć testy zabezpieczeń. Inną opcją będzie dodanie skrótu złośliwego pliku do bazy danych `db` i usunięcie go z `dbx` (gdy skrót ten znajdował się pierwotnie w bazie danych `dbx`). Jak widać na rysunku 17.10, zmieniając zawartość zmiennej `PK`, aby zawierała certyfikat klucza publicznego napastnika, może on następnie dodawać i usuwać klucze wymiany kluczy ze zmiennej `KEK` UEFI, co z kolei zapewni mu kontrolę nad bazami podpisów `db` i `dbx`, łamiąc ochronę Secure Boot.



Rysunek 17.10. Atak przeciwko łańcuchowi zaufania UEFI Secure Boot

Trzecia opcja, zamiast zmieniania klucza platformy i przejmowania leżącej w tle hierarchii PKI, polega po prostu na uszkodzeniu klucza platformy w zmiennej UEFI. Aby móc działać, Secure Boot potrzebuje poprawnego klucza platformy wystawionego dla oprogramowania układowego platformy; w przeciwnym razie ochrona jest wyłączona.

Czytelnicy zainteresowani poznaniem większej liczby szczegółów na temat takich ataków powinni zapoznać się z następującymi materiałami konferencyjnymi, które zawierają wyczerpujące analizy technologii UEFI Secure Boot:

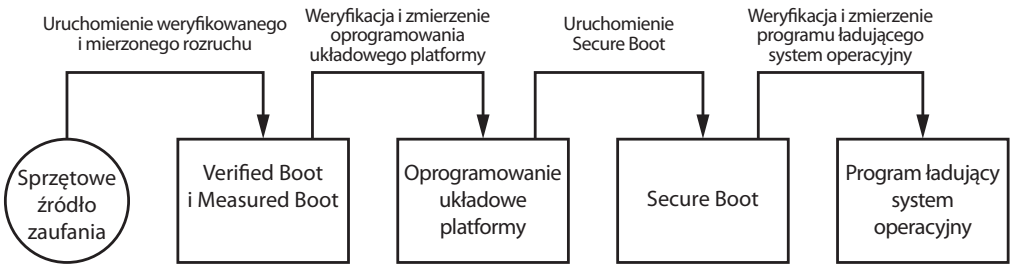
- Corey Kallenberg i in. „Setup for failure: defeating secure boot", LegbaCore, *https://papers.put.as/papers/firmware/2014/SetupForFailure -syscan-v4.pdf*.
- Yuriy Bulygin i in. „Summary of attacks against BIOS and Secure Boot", Intel Security, *http://www.c7zero.info/stuff/DEFCON22 -BIOSAttacks.pdf*.

Ochrona Secure Boot za pomocą weryfikowanego lub mierzonego rozruchu

Jak widzimy, sam Secure Boot nie jest w stanie uchronić przed atakami, które obejmują zmiany w oprogramowaniu układowym platformy. Czy istnieje zatem jakaś ochrona dla samej technologii Secure Boot? Odpowiedź brzmi: tak. W tym podrozdziale zajmiemy się technologiami zabezpieczeń mających na celu chronić systemowe oprogramowanie układowe przed nieautoryzowanymi modyfikacjami – a konkretnie weryfikowanym rozruchem (*Verified Boot*) oraz mierzonym rozruchem (*Measured Boot*). Verified Boot oznacza sprawdzenie, że oprogramowanie układowe platformy nie zostało

zamienione ani zmodyfikowane, podczas gdy Measured Boot oblicza skróty kryptograficzne określonych komponentów uczestniczących w procesie rozruchu i przechowuje je w rejestrach konfiguracji platformy TPM (*Trusted Platform Module Platform Configuration Register* – TPM PCR).

Metody te są niezależne od siebie i możliwa jest sytuacja, gdy dana platforma ma włączoną tylko jedną z tych funkcji lub obie. Niemniej jednak oba rozwiązania stanowią części tego samego łańcucha zaufania (pokazanego na rysunku 17.11).



Rysunek 17.11. Przepływ operacji w Verified Boot i Measured Boot

Jak widzieliśmy na rysunku 17.8, oprogramowanie układowe inicjowania platformy jest fragmentem kodu wykonywanym jako pierwszy po wyjściu procesora ze stanu resetu. UEFI Secure Boot bezwarunkowo ufa oprogramowaniu inicjowania platformy, zatem sensowne jest oparcie faktycznych ataków przeciwko Secure Boot na nieautoryzowanych modyfikacjach tego oprogramowania.

Aby ochronić się przed takimi atakami, system potrzebuje źródła zaufania na zewnątrz oprogramowania układowego inicjowania platformy. W tym miejscu do gry wkracza Verified Boot i Measured Boot. Procesy te wykonują mechanizmy zabezpieczeń, których źródło zaufania jest zakotwiczone w sprzęcie. Co więcej, są wykonywane przed oprogramowaniem układowym, co oznacza, że są w stanie zarówno je uwierzytelnić, jak i pomierzyć. Tym, co w tym kontekście oznacza mierzenie, zajmiemy się za chwilę.

Weryfikowany rozruch

Po włączeniu zasilania systemu wyposażonego w Verified Boot sprzętowa logika uruchamia funkcjonalność weryfikacji rozruchowej implementowanej w pamięci ROM lub mikrokodzie wewnątrz procesora. Logika ta jest *niezmienna*, co oznacza, że nie można jej zmienić programowo. Zazwyczaj Verified Boot wykonuje moduł weryfikujący integralność systemu, aby zapewnić, że system ten wykona autentyczne oprogramowanie układowe bez złośliwych modyfikacji. Na potrzeby weryfikowania oprogramowania układowego wykorzystywana jest kryptografia klucza publicznego; podobnie jak

UEFI Secure Boot: sprawdzany jest podpis cyfrowy oprogramowania platformy, aby zagwarantować jego autentyczność. Po udanym uwierzytelnieniu uruchamiane jest oprogramowanie układowe platformy, które przechodzi do weryfikowania innych komponentów (np. układów Option ROM, sterowników DXE i programów ładujących system operacyjny), aby utrzymać właściwy łańcuch zaufania. Na tym polega część weryfikacyjna rozwiązań Verified Boot i Measured Boot.

Mierzony rozruch

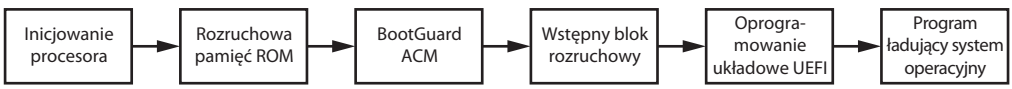
Mechanizm Measured Boot polega na „mierzeniu" oprogramowania układowego platformy i programów ładujących system operacyjny. Oznacza to, że mechanizm ten oblicza skróty kryptograficzne komponentów uczestniczących w procesie rozruchu. Skróty te są przechowywane w zbiorze rejestrów TPM PCR. Same wartości skrótów nie mówią nic o tym, czy badane komponenty są dobre, czy złośliwe, ale informują, czy komponenty konfiguracji i rozruchu zostały zmienione w jakimś miejscu. Jeśli komponent rozruchowy został zmodyfikowany, jego wartość skrótu będzie się różnić od tej obliczonej dla wersji oryginalnej. Tym samym mierzenie rozruchu pozwala zauważyć dowolną modyfikację komponentu rozruchu.

Później oprogramowanie systemowe może użyć skrótów zawartych w tych TPM PCR do zagwarantowania, że system działa w znanym dobrym stanie bez żadnych podstępnych modyfikacji. System może również użyć tych skrótów do zdalnej atestacji, w której system próbuje udowodnić innemu systemowi, że znajduje się w stanie zaufanym.

Teraz, gdy znamy już ogólne działanie mechanizmów rozruchu, przyjrzyjmy się ich kilku implementacjom, zaczynając od Intel BootGuard.

Intel BootGuard

Intel BootGuard to opracowana przez firmę Intel technologia weryfikowanego i mierzonego rozruchu. Na rysunku 17.12 jest pokazany przepływ operacji na platformie z włączoną funkcjonalnością BootGuard.



Rysunek 17.12. Przepływ działań Intel BootGuard

W trakcie inicjowania, zanim procesor rozpocznie wykonywanie pierwszego kodu zlokalizowanego w wektorze resetu, wykonuje kod z rozruchowej pamięci ROM. Kod ten wykonuje niezbędną operację inicowania stanu procesora, po czym ładuje i wykonuje BootGuard *Authenticated Code Module* (ACM).

ACM jest modułem specjalnego typu służącym do wykonywania operacji wrażliwych ze względów zabezpieczeń i musi być podpisany przez firmę Intel. Tak więc kod rozruchowego ROM, który ładuje ACM, wykonuje obowiązkową weryfikację podpisu, aby powstrzymać moduł przed uruchomieniem czegokolwiek, co nie zostało podpisane przez firmę Intel. Po udanej weryfikacji podpisu ACM jest wykonywany w izolowanym środowisku, aby powstrzymać dowolne złośliwe oprogramowanie przed wpływaniem na jego działanie.

BootGuard ACM implementuje funkcjonalność Verified i Measured Boot. Moduł ten ładuje do pamięci loader oprogramowania układowego pierwszej fazy, nazywany wstępnym blokiem rozruchowym (*initial boot block* – IBB), po czym, zależnie od stosowanej zasady rozruchu, weryfikuje go i/lub mierzy. IBB stanowi część oprogramowania układowego zawierającego kod, który jest wykonywany po resecie.

Mówiąc ściślej, w tym momencie procesu rozruchu nie ma jeszcze pamięci RAM. Kontroler pamięci nie został jeszcze zainicjowany, zatem RAM nie jest dostępny. Niemniej jednak procesor konfiguruje swoją pamięć podręczną (*cache*) ostatniego poziomu, aby mogła być używana jako RAM, wprowadzając ją w tryb Cache-as-RAM do tego punktu w procesie rozruchu, w którym kod BIOS zdoła skonfigurować kontroler pamięci i odkryć pamięć RAM.

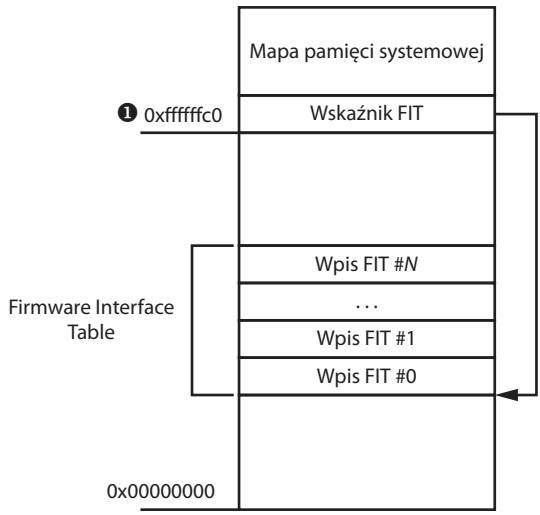
ACM przekazuje sterowanie do IBB, gdy IBB zostanie z powodzeniem zweryfikowany i/lub zmierzony. Jeśli weryfikacja IBB się nie powiedzie, zachowanie ACM zależy od aktywnej zasady rozruchu: system może wykonać natychmiastowe zamknięcie albo pozwolić na przywracanie oprogramowania układowego po pewnym czasie.

Następnie IBB ładuje resztę oprogramowania układowego UEFI z pamięci flash SPI i je weryfikuje i/lub mierzy. Gdy już IBB przejmie sterowanie, Intel BootGuard nie jest dłużej odpowiedzialny za utrzymanie właściwego łańcuchu zaufania, gdyż jego przeznaczeniem jest jedynie weryfikacja i badanie IBB. To IBB jest odpowiedzialne za kontynuowanie łańcucha zaufania do momentu, w którym UEFI Secure Boot przejmuje zadania weryfikowania i mierzenia obrazów oprogramowania układowego.

Odszukiwanie ACM

Przyjrzyjmy się szczegółom implementacji technologii Intel BootGuard dla platform biurkowych, poczynając od ACM. Ponieważ ACM jest jednym z pierwszych komponentów Intel BootGuard wykonywanych po włączeniu zasilania, pierwsze pytanie brzmi: jak procesor odszukuje ACM, gdy zostanie włączony?

Dokładna lokalizacja ACM udostępniana jest w specjalnej strukturze danych o nazwie *Firmware Interface Table (FIT)* (tabela interfejsów oprogramowania układowego), przechowywanej w obrazie oprogramowania układowego. FIT jest uporządkowana jako tablica wpisów FIT, z których każdy opisuje lokalizację w oprogramowaniu określonego obiektu, takiego jak ACM czy pliki aktualizacji mikrokodu. Na rysunku 17.13 jest pokazany układ FIT w pamięci systemowej po resecie.



Rysunek 17.13. Lokalizacja FIT w pamięci

Po włączeniu zasilania procesor odczytuje adres FIT z lokalizacji pamięci 0xFFFFFFC0 ❶. Ponieważ jeszcze nie ma pamięci RAM, gdy procesor zgłasza transakcję odczytu pamięci spod adresu fizycznego 0xFFFFFFC0, wewnętrzna logika chipsetu rozpoznaje, że adres ten należy do specjalnego zakresu i zamiast wysłania tej transakcji do kontrolera pamięci, dekoduje ją. Transakcje odczytu pamięci dotyczące tabeli FIT są przekierowywane do kontrolera flash SPI, który odczytuje FIT z pamięci flash.

Przyjrzymy się bliżej temu procesowi, wracając do repozytorium EDK 2. W katalogu *IntelSiliconPkg/Include/IndustryStandard/* znajdziemy plik nagłówkowy *FirmwareInterfaceTable.h*, który zawiera pewne definicje kodu odpowiadające strukturze FIT. Układ wpisów FIT jest pokazany w listingu 17.2.

```
typedef struct {
  UINT64 Address; ❶
  UINT8 Size[3]; ❷
  UINT8 Reserved;
  UINT16 Version; ❸
  UINT8 Type : 7; ❹
  UINT8 C_V : 1; ❺
  UINT8 Chksum; ❻
} FIRMWARE_INTERFACE_TABLE_ENTRY;
```

Listing 17.2. Układ wpisów FIT

Jak wspomnieliśmy, każdy wpis tabeli FIT opisuje określony obiekt w obrazie oprogramowania układowego. Natura każdego obiektu jest zakodowana w polu `Type` struktury FIT. Obiekty te mogą być na przykład plikiem aktualizacji mikrokodu, blokiem ACM mechanizmu BootGuard albo zasadą BootGuard. Pola `Address` ❶ oraz `Size` ❷ udostępniają lokalizację obiektu w pamięci: `Address` zawiera fizyczny adres obiektu, a `Size` definiuje rozmiar wyrażony w jednostkach `dword` (wartości 4 -bajtowe). Pole `C_V` ❺ oznacza poprawna suma kontrolna (*checksum valid*); jeśli jest ustawiona na 1, pole `Chksum` ❻ zawiera poprawną sumę kontrolną obiektu. Suma wszystkich bajtów komponentu modulo 0xFF oraz wartości w polu `Chksum` musi być równa zeru. Pole `Version` ❸ zawiera numer wersji komponentu w formacie BCD. Wartość w polu wpisu nagłówka FIT wskazuje numer rewizji struktury danych FIT.

Plik nagłówkowy *FirmwareInterfaceTable.h* zawiera wartości, które może przyjmować pole `Type` ❹. Te wartości są w większości nieudokumentowane i dostępnych jest niewiele informacji, ale definicje typów wpisów FIT są dość szczegółowe i z kontekstu można wydedukować ich znaczenie. Oto typy istotne dla BootGuard:

- Wpis `FIT_TYPE_00_HEADER` zawiera całkowitą liczbę wpisów FIT w tabeli FIT w jego polu `Size`. Jego pole adresu zawiera specjalną ośmiobajtową sygnagurę `'_FIT_ '` (mamy tu trzy spacje po `_FIT_`).
- Wpis typu `FIT_TYPE_02_STARTUP_ACM` udostępnia lokalizację modułu ACM BootGuard, którą przetwarza kod rozruchowego ROM w celu zlokalizowania ACM w pamięci systemowej.
- Wpisy o typach `FIT_TYPE_0C_BOOT_POLICY_MANIFEST` (manifest zasady rozruchu BootGuard) oraz `FIT_TYPE_0B_KEY_MANIFEST` (manifest klucza BootGuard) udostępniają mechanizmowi BootGuard aktywną zasadę rozruchu oraz informacje konfiguracyjne, które omówimy pokrótce w podpunkcie „Konfigurowanie Intel BootGuard” na stronie 384.

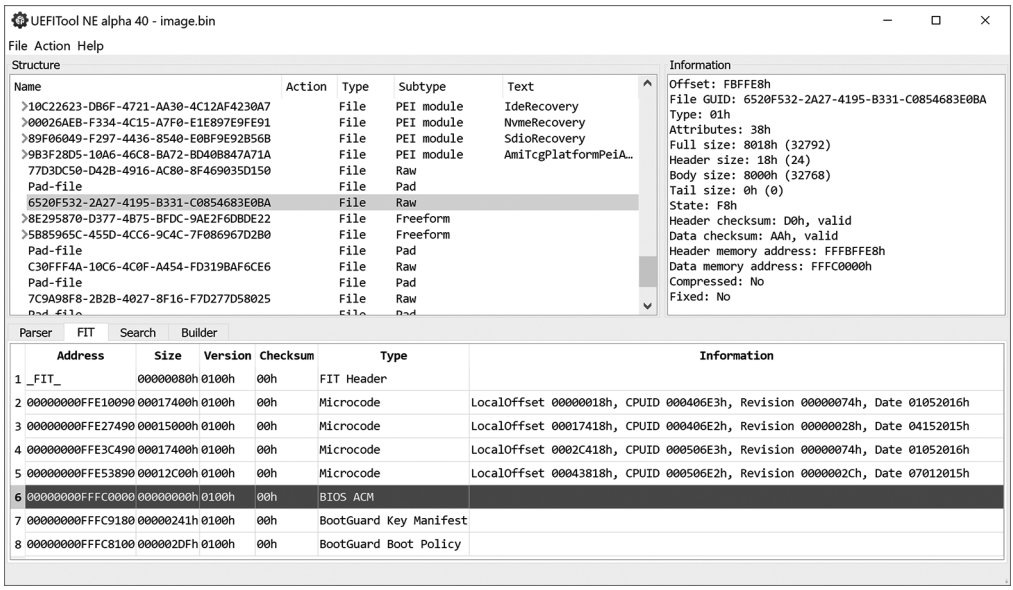
Trzeba mieć na uwadze, że zasada rozruchu Intel BootGuard i zasada UEFI Secure Boot to dwie różne rzeczy. Pierwszy termin odnosi się do zasady rozruchu używanej w procedurach Verified i Measured Boot. Innymi słowy, zasada rozruchu Intel BootGuard jest wymuszana przez ACM i logikę chipsetu i zawiera takie parametry, jak to, czy BootGuard ma wykonywać badanie rozruchu oraz co powinien zrobić w sytuacji, gdy nie uda się uwierzytelnienie IBB. Drugi pojęcie dotyczy UEFI Secure Boot, omówionego wcześniej w tym rozdziale, i jest w całości wymuszana przez oprogramowanie układowe UEFI.

Poznawanie FIT

Niektóre wpisy FIT w obrazie oprogramowania układowego można poznać przy użyciu narzędzia UEFITool, które przedstawiliśmy w rozdziale 15 (i do

którego powrócimy jeszcze w rozdziale 19), a także wydobyć z obrazu ACM wraz z manifestami zasady rozruchu oraz klucza do dalszej analizy. Może to być użyteczne, gdyż ACM może zostać wykorzystany do ukrycia złośliwego kodu. W poniższym przykładzie użyjemy obrazu oprogramowania układowego uzyskanego z systemu z włączoną technologią Intel BootGuard. (Rozdział 19 zawiera informacje, jak wydobyć oprogramowanie układowe z platformy).

Najpierw ładujemy obraz oprogramowania układowego do narzędzia UEFITool, wybierając polecenie **File ▸ Open Image File**. Po wskazaniu pliku obrazu do załadowania zobaczymy okno podobne do przedstawionego na rysunku 17.14.



Rysunek 17.14. Przeglądanie FIT w programie UEFITool

W dolnej części okna możemy zobaczyć kartę FIT, która wylicza wpisy. W kolumnie Type tej karty są wyświetlone typy wpisów FIT. Szukamy wpisów FIT dla typów BIOS ACM, BootGuard Key Manifest oraz BootGuard Boot Policy. Używając tych informacji, możemy zlokalizować komponenty Intel BootGuard w obrazie oprogramowania układowego i wydobyć je do dalszej analizy. W tym konkretnym przykładzie wpis FIT o numerze 6 wskazuje lokalizację BIOS ACM; rozpoczyna się ona od adresu 0xfffc0000. Wpisy FIT o numerach 7 i 8 wskazują lokalizacje manifestów klucza i zasady rozruchu; rozpoczynają się one odpowiednio od adresów 0xfffc9180 i 0xfffc8100.

Konfigurowanie Intel BootGuard

W trakcie wykonywania BootGuard BIOS ACM konsumuje klucz BootGuard, podczas gdy zasada rozruchu lokalizuje IBB w pamięci, aby uzyskać poprawny klucz publiczny w celu zweryfikowania podpisu IBB.

Manifest klucza BootGuard zawiera skrót manifestu zasady rozruchu (*boot policy manifest* – BPM), główny klucz publiczny OEM, podpis cyfrowy poprzednich pól (z wyjątkiem głównego klucza publicznego, który nie jest uwzględniony w podpisanych danych) oraz numer wersji zabezpieczeń (licznik inkrementowany przy każdej aktualizacji zabezpieczeń w celu uniemożliwienia ataków przez odwołanie aktualizacji).

Sam BPM zawiera numer wersji zabezpieczeń, lokalizację oraz skrót IBB, klucz publiczny BPM oraz cyfrowe podpisy wymienionych właśnie pól BPM – ponownie z wyjątkiem głównego klucza publicznego, który można walidować przy użyciu klucza publicznego BPM. Lokalizacja IBB udostępnia układ IBB w pamięci. Nie musi to być ciągły blok pamięci; może się składać z kilku niepowiązanych regionów pamięci. Skrót IBB zawiera skumulowaną wartość skrótu dla wszystkich obszarów pamięci zajmowanych przez IBB. Tym samym cały proces weryfikowania podpisu IBB przebiega następująco:

(1) BootGuard lokalizuje manifest klucza (KM) przy użyciu FIT i uzyskuje wartość skrótu manifestu zasady rozruchu oraz główny klucz OEM (*root key*), który będziemy nazywali kluczem 1. BootGuard weryfikuje podpis cyfrowy w KM, używając klucza 1, aby zagwarantować integralność wartości skrótu BPM. Jeśli weryfikacja się nie powiedzie, BootGuard zgłasza błąd i wywołuje działania naprawcze.

(2) Jeśli weryfikacja się powiedzie, BootGuard lokalizuje BPM przy użyciu FIT, oblicza wartość skrótu dla BPM i porównuje ją ze skrótem BPM zawartym w KM. Jeśli wartości nie są równe, BootGuard zgłasza błąd i wywołuje działania naprawcze; w przeciwnym razie odczytuje z BPM wartość skrótu i lokalizację IBB.

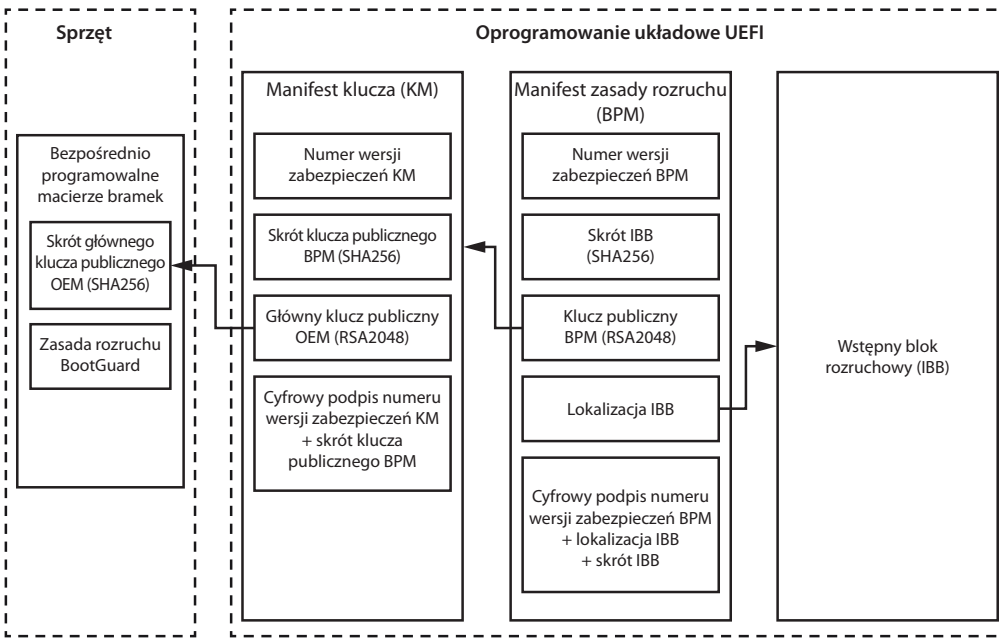
(3) BootGuard lokalizuje IBB w pamięci, oblicza jego skumulowany skrót i porównuje go z wartością skrótu przechowywaną w BPM. Jeśli skróty nie są równe, BootGuard zgłasza błąd i wywołuje działania naprawcze.

(4) W przeciwnym razie BootGuard zgłasza, że weryfikacja się powiodła. Jeśli włączony jest mierzony rozruch, BootGuard wykonuje również pomiar IBB przez obliczenie jego skrótu i zapisuje miarę w TPM. Następnie BootGuard przekazuje sterowanie do IBB.

KM jest najistotniejszą strukturą, ponieważ zawiera główny klucz publiczny OEM wykorzystywany do weryfikacji integralności IBB. Można postawić pytanie: „Jeśli KM BootGuard przechowywane jest w niechronionym układzie flash SPI wraz z obrazem oprogramowania układowego, czy nie oznacza to, że napastnik mógłby zmodyfikować go w pamięci flash, aby dostarczyć do BootGuard fałszywy klucz weryfikacyjny?”. Aby uniemożliwić

taki atak, skrót głównego klucza publicznego OEM jest zamiast tego przechowywany w bezpośrednio programowalnych macierzach bramek w chipsecie. Macierze te można zaprogramować tylko raz, w chwili wprowadzania zasady rozruchu BootGuard. Gdy macierz zostanie zapisana, nie można już jej nadpisać. W ten właśnie sposób klucz weryfikacji BootGuard jest umocowany w sprzęcie, czyniąc go niezmiennym źródłem zaufania. (Również zasada rozruchu BootGuard jest zapisana w macierzach bramek chipsetu, przez co nie jest możliwa zmiana zasady po jej ustawieniu).

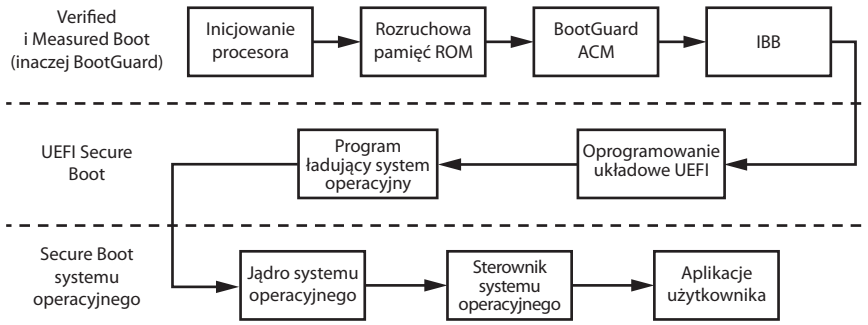
Jeżeli napastnik zmieni manifest klucza BootGuard, ACM zauważy przeróbkę klucza, obliczając jego skrót i porównując ją ze „złotą" wartością wtopioną w chipset. Niezgodne skróty wyzwolą zgłoszenie błędu i działanie naprawcze. Na rysunku 17.15 jest pokazany łańcuch zaufania wymuszany przez BootGuard.



Rysunek 17.15. Łańcuch zaufania Intel BootGuard

Gdy IBB zostanie zweryfikowany z powodzeniem i, jeśli to wymagane, zmierzony, zostaje uruchomiony i wykonuje pewne podstawowe inicjowanie chipsetu, po czym ładuje oprogramowanie układowe UEFI. W tym momencie to IBB odpowiada za uwierzytelnienie oprogramowania UEFI przed jego załadowaniem i uruchomieniem. W przeciwnym razie łańcuch zaufania zostanie zerwany.

Rysunek 17.16 podsumowuje ten podrozdział, prezentując granice odpowiedzialności w implementacjach Secure Boot.



Rysunek 17.16. Granice odpowiedzialności w implementacji Secure Boot

ARM Trusted Boot Board

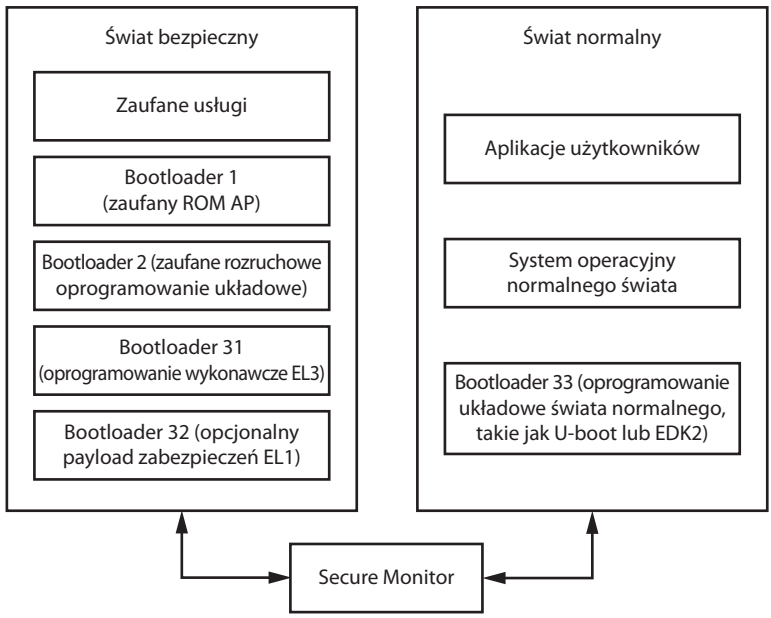
Architektura ARM dysponuje własną implementacją technologii weryfikowanego i mierzonego rozruchu o nazwie *Trusted Boot Board (TBB)* albo po prostu *Trusted Boot* (zaufany rozruch). W tym podrozdziale przyjrzymy się projektowi Trusted Boot. ARM wykorzystuje bardzo szczególną konfigurację, znaną pod nazwą technologii zabezpieczeń *Trust Zone* (strefa zaufana), która dzieli środowisko wykonawcze na dwie części. Zanim zagłębimy się w detale procesu weryfikowanego i mierzonego rozruchu używanego w ARM, musimy wyjaśnić, jak działają strefy zaufania.

ARM Trust Zone

Technologia zabezpieczeń Trust Zone to implementowana sprzętowo funkcjonalność, która dzieli środowisko wykonawcze ARM na dwa „światy": świat bezpieczny (*secure world*) oraz normalny (*niezabezpieczony*), które współistnieją w tym samym rdzeniu fizycznym, co pokazano na rysunku 17.17. Logika implementowana w sprzęcie procesora i oprogramowaniu układowym gwarantuje, że zasoby świata bezpiecznego są właściwie odizolowane i chronione przed oprogramowaniem działającym w świecie niezabezpieczonym.

Oba „światy" mają własne, dedykowane i oddzielne stosy oprogramowania układowego i zwykłego: w normalnym świecie działają aplikacje użytkowników i system operacyjny, podczas gdy w świecie bezpiecznym działa zabezpieczony system operacyjny i zaufane usługi. Oprogramowanie układowe tych światów składa się z różnych programów ładujących (*bootloader*), odpowiedzialnych za inicjowanie świata i ładowanie systemu operacyjnego, którymi zajmiemy się za chwilę. Z tego powodu bezpieczny i normalny świat mają różne obrazy oprogramowania układowego.

Wewnątrz procesora oprogramowanie działające w normalnym świecie nie może bezpośrednio uzyskać dostępu do kodu i danych w świecie bezpiecznym. Logika kontrolująca dostęp, która to uniemożliwia, jest implementowa-



Rysunek 17.17. ARM Trust Zone

na sprzętowo, zazwyczaj w sprzęcie zwanym *System on Chip*. Niemniej jednak oprogramowanie działające w świecie normalnym może przekazać sterowanie do oprogramowania w świecie bezpiecznym (np. w celu wykonania zaufanej usługi w świecie bezpiecznym), wykorzystując specjalne oprogramowanie zwane monitorem bezpieczeństwa (*Secure Monitor*; w procesorach ARM Cortex-A) albo logiką rdzenia (w procesorach ARM Cortex-M). Mechanizm ten gwarantuje, że przełączanie się między światami nie naruszy zabezpieczeń systemu.

Łącznie technologie Trusted Boot oraz Trust Zone tworzą zaufane środowisko wykonawcze (*Trusted Execution Environment*) służące do uruchamiania oprogramowania o wysokich uprawnieniach i zapewniania środowiska dla takich technologii zabezpieczeń, jak zarządzanie prawami cyfrowymi, algorytmy kryptograficzne i uwierzytelniania oraz inne zastosowania wrażliwe pod względem zabezpieczeń. W ten sposób izolowane, chronione środowisko może utrzymywać najbardziej wrażliwe oprogramowanie.

Programy ładujące w architekturze ARM

Ponieważ świat bezpieczny i normalny są utrzymywane w separacji, każdy potrzebuje własnego zestawu programów ładujących. Dodatkowo proces rozruchu każdego świata składa się z wielu etapów, co oznacza, że wiele programów ładujących musi zostać uruchomionych w różnych momentach procesu rozruchu. W tym podrozdziale ogólnie opiszemy przebieg Trusted

Boot dla procesorów aplikacji ARM, zaczynając od poniższej listy programów ładujących (*bootloader*) uczestniczących w tym procesie. Pokazaliśmy je już wcześniej na rysunku 17.17.

BL1 – program ładujący pierwszego etapu, zlokalizowany w rozruchowej pamięci ROM i wykonywany w świecie bezpiecznym;

BL2 – program ładujący drugiego etapu, zlokalizowany w pamięci flash, ładowany i uruchamiany przez BL1 w świecie bezpiecznym;

BL31 – oprogramowanie układowe czasu działania dla świata bezpiecznego, ładowane i uruchamiane przez BL2;

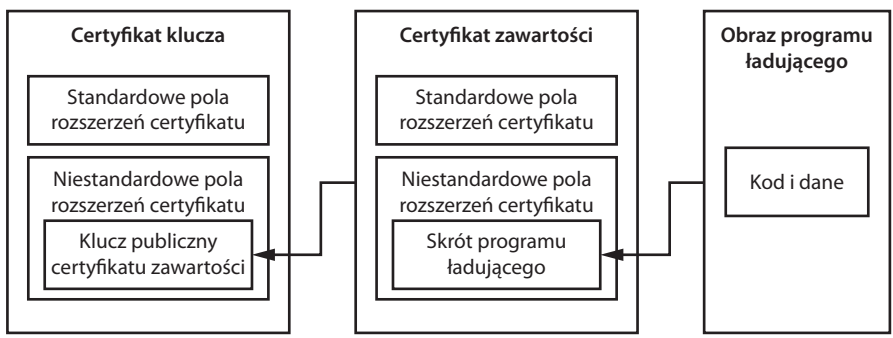
BL32 – opcjonalny program ładujący trzeciego etapu świata bezpiecznego, ładowany przez BL2;

BL33 – oprogramowanie układowe czasu działania dla świata normalnego, ładowane i uruchamiane przez BL2.

Nie jest to pełna ani dokładna lista wszystkich implementacji ARM występujących w praktyce, gdyż niektórzy producenci wprowadzają dodatkowe programy ładujące albo usuwają niektóre z istniejących. W niektórych przypadkach BL1 może nie być pierwszym kodem wykonywanym w procesorze aplikacji, gdy system powraca do działania po resecie.

W celu weryfikacji integralności tych komponentów rozruchu Trusted Boot opiera się na certyfikatach klucza publicznego standardu X.509 (przypomnijmy, że pliki w bazie danych `db` UEFI Secure Boot też były zakodowane przy użyciu X.509). Warto wspomnieć, że wszystkie certyfikaty są samo-podpisane. Nie ma tu potrzeby tworzenia urzędu certyfikacji, gdyż łańcuch zaufania nie jest tworzony przez ważność wystawcy certyfikatu, ale przy użyciu zawartości rozszerzeń certyfikatów.

Trusted Boot używa dwóch typów certyfikatów: klucza i zawartości. Certyfikatów klucza używa najpierw do weryfikowania kluczy publicznych użytych do podpisania certyfikatów zawartości. Następnie wykorzystuje certyfikaty zawartości do przechowania skrótów obrazów programów ładujących. Zależność ta została przedstawiona na rysunku 17.18.



Rysunek 17.18. Certyfikaty klucza i zawartości Trusted Boot

Trusted Boot uwierzytelnia obraz, obliczając jego skrót i porównując wynik ze skrótem wydobytym z certyfikatu zawartości.

Przebieg Trusted Boot

Po zapoznaniu się z podstawowymi koncepcjami Trusted Boot przyjrzyjmy się przebiegowi tego procesu dla procesora aplikacji, pokazanego na rysunku 17.19. Powinno dać to nam pełny obraz tego, jak w procesorach ARM implementowane są rozwiązania weryfikowanego rozruchu i jak chroni to platformę przed wykonaniem niezaufanego kodu, w tym rootkitów oprogramowania układowego.

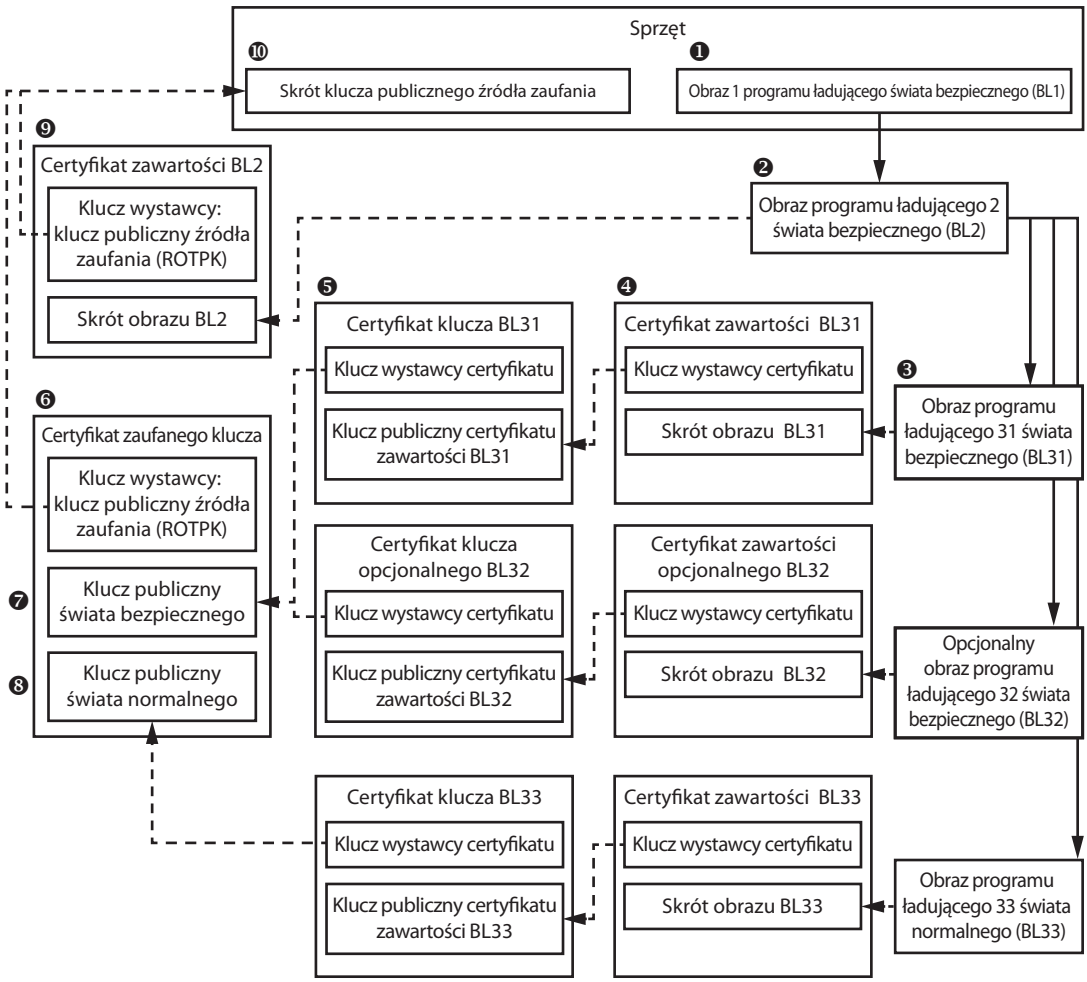
Wypełnione strzałki na rysunku 17.19 oznaczają przekazywanie przepływu wykonywania, a przerywane – relacje zaufania. Innymi słowy, każdy element ufa takiemu, który wskazuje wychodząca zeń przerywana strzałka.

Gdy procesor wyprowadzany jest ze stanu resetu, pierwszym wykonywanym fragmentem kodu jest program ładujący 1 (BL1) ❶. BL1 ładowany jest z rozruchowej pamięci ROM tylko do odczytu, co oznacza, że nie może zostać zmodyfikowany, gdy już zostanie w niej umieszczony. BL1 wczytuje certyfikat zawartości programu ładującego 2 (BL2) ❾ z pamięci flash i sprawdza jego klucz wystawcy. BL1 oblicza następnie skrót certyfikatu zawartości wystawcy BL2 i porównuje go ze „złotą" wartością przechowywaną w zabezpieczonym *źródłowym rejestrze kluczy publicznych* (*root of trust public key register* – ROTPK) w sprzęcie ❿. Rejestr ROTPK oraz rozruchowy ROM są zamocowanymi w sprzęcie źródłami zaufania dla Trusted Boot. Jeśli skróty nie są równe albo weryfikacja podpisu certyfikatu zawartości BL2 się nie powiedzie, system przechodzi w stan awaryjny.

Gdy certyfikat zawartości BL2 zostanie zweryfikowany względem ROTPK, BL1 ładuje obraz BL2 z pamięci flash ❷, oblicza jego skrót kryptograficzny i porównuje ten skrót z wartością wydobytą z certyfikatu zawartości BL2 ❺.

Po uwierzytelnieniu BL1 przekazuje sterowanie do BL2, które z kolei odczytuje swój certyfikat zaufanego klucza ❻ z pamięci flash. Ten certyfikat zawiera klucze publiczne na potrzeby weryfikacji oprogramowania układowego dla świata bezpiecznego ❼ i normalnego ❽. Klucz wystawcy certyfikatu zaufanych kluczy jest sprawdzany względem rejestru ROTPK ❿.

Następnie BL2 uwierzytelnia BL31 ❸, który stanowi oprogramowanie układowe czasu działania dla świata bezpiecznego. Aby uwierzytelnić obraz BL31, BL2 używa certyfikatów klucza i zawartości dla BL31 ❹. BL2 weryfikuje te certyfikaty kluczy, używając klucza publicznego świata bezpiecznego uzyskanego z certyfikatu zaufanego klucza. Certyfikat klucza BL31 zawiera klucz publiczny certyfikatu zawartości BL31 służący do weryfikacji podpisu certyfikatu zawartości BL32.



Rysunek 17.19. Przebieg Trusted Boot

Po zweryfikowaniu certyfikatu zawartości BL31 wartość skrótu obrazu BL31 przechowywana w tym certyfikacie BL31 jest używana do sprawdzenia integralności obrazu BL31. Ponownie dowolne niepowodzenie powoduje stan awaryjny systemu.

Analogicznie BL2 sprawdza integralność opcjonalnego obrazu BL32 świata bezpiecznego, używając certyfikatów klucza i zawartości BL32.

Integralność obrazu oprogramowania układowego BL33 (wykonywanego w świecie normalnym) jest sprawdzana za pomocą certyfikatów klucza i zawartości BL33. Certyfikat klucza BL33 jest weryfikowany przy użyciu klucza publicznego świata normalnego uzyskanego z certyfikatu zaufanego klucza.

Jeśli wszystkie testy przebiegną z powodzeniem, system przechodzi do wykonania uwierzytelnionego oprogramowania układowego dla obu światów – bezpiecznego i normalnego.

> **Hardware Validated Boot firmy AMD**
>
> Choć nie omawiamy jej w tym rozdziale, firma AMD opracowała własną implementację weryfikowanego i mierzonego rozruchu, nazywaną Hardware Validated Boot (HVB). Technologia ta implementuje funkcjonalność analogiczną do Intel BootGuard. Oparta na technologii AMD Platform Security Processor zawiera dedykowany mikrokontroler operacji związanych z zabezpieczeniami, który działa niezależnie od głównego rdzenia systemowego.

Weryfikowany rozruch kontra rootkity oprogramowania układowego

Dysponując całą tą wiedzą, możemy ostatecznie stwierdzić, czy weryfikowany rozruch może ochronić przed rootkitami oprogramowania układowego.

Wiemy, że weryfikowany rozruch następuje przed wykonaniem jakiegokolwiek oprogramowania układowego w procesie rozruchu. Oznacza to, że gdy mechanizm ten rozpoczyna weryfikację, żaden rootkit infekujący oprogramowanie układowe nie będzie jeszcze aktywny, zatem złośliwe oprogramowanie nie może przeciwdziałać procesowi weryfikacji. Weryfikowany rozruch wykryje dowolną podstępną modyfikację oprogramowania układowego i uniemożliwi jej wykonanie.

Co więcej, źródło zaufania weryfikowanego rozruchu jest umocowane w sprzęcie, zatem napastnicy nie mogą go zniekształcić. Główny klucz publiczny OEM w rozwiązaniu Intel BootGuard jest „zaszyty" w chipsecie, a klucz źródła zaufania ARM jest przechowywany w zabezpieczonych rejestrach. W obu przypadkach kod startowy rozpoczynający Verified Boot jest ładowany z pamięci tylko do odczytu, więc złośliwe oprogramowanie nie może go łatać ani modyfikować.

Możemy zatem podsumować, że Verfied Boot może wytrzymać ataki ze strony rootkitów oprogramowania układowego. Jak jednak można było zauważyć, cała ta technologia jest dość złożona. Ma wiele zależności, zatem łatwo może się zdarzyć, że zostanie nieprawidłowo zaimplementowana. Technologia ta jest tylko tak bezpieczna, jak jej najsłabszy komponent; pojedynczy defekt w łańcuchu zaufania sprawia, że możliwe jest jego ominięcie. Oznacza to, że istnieje spora szansa na to, aby napastnicy znaleźli podatności w implementacji Verified Boot, które można wykorzystać i zainstalować rootkity oprogramowania układowego.

Podsumowanie

W tym rozdziale zbadaliśmy trzy technologie Secure Boot: UEFI Secure Boot, Intel BootGuard oraz ARM Trusted Boot. Wszystkie te technologie polegają na istnieniu łańcucha zaufania – wymuszanego od samego początku procesu rozruchu aż po wykonanie aplikacji użytkownika – i angażują wielką liczbę modułów rozruchowych. Gdy zostaną poprawnie zaimplementowane i skonfigurowane, zapewniają dobrą ochronę przed ciągle rosnącą liczbą rootkitów oprogramowania układowego UEFI. To dlatego systemy o wysokich wymaganiach pewności muszą używać Secure Boot i dlatego obecnie wiele systemów konsumenckich ma domyślnie włączony ten mechanizm. W kolejnym rozdziale skupimy się na metodach analizowania rootkitów oprogramowania układowego.

18

SPOSOBY ANALIZOWANIA UKRYTYCH SYSTEMÓW PLIKÓW

Do tej pory zajmowaliśmy się tym, jak bootkity penetrują i utrwalają swoją obecność w komputerze ofiary, używając wyrafinowanych technik w celu uniknięcia wykrycia. Jedną ze wspólnych cech takich zaawansowanych zagrożeń jest wykorzystywanie niestandardowego, ukrytego systemu plików do przechowywania modułów i informacji konfiguracyjnych na przejętej maszynie.

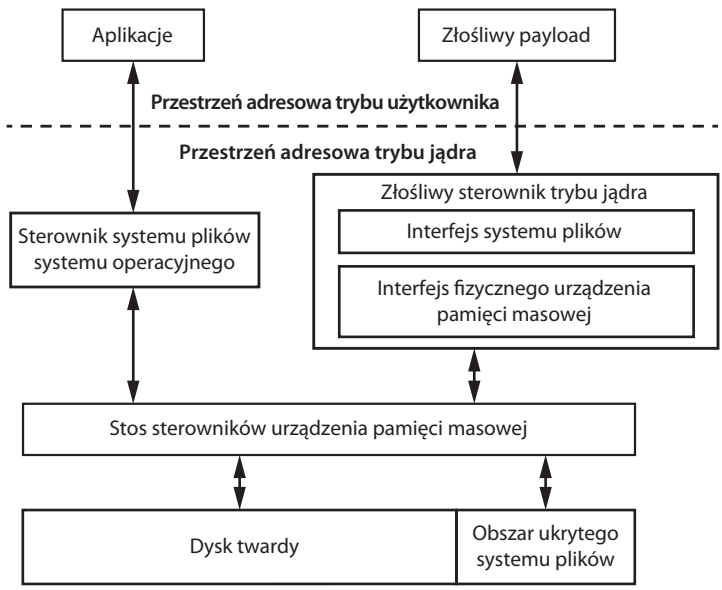
Wiele ukrytych systemów plików występujących w złośliwym oprogramowaniu to rozwiązania własne lub przerobione wersje standardowych systemów, co oznacza, że wykonywanie analizy na komputerze, który został przejęty przez rootkit lub bootkit, często wymaga niestandardowego zestawu narzędzi. Aby móc opracować takie narzędzia, badacze muszą poznać układ ukrytego systemu plików i algorytmy użyte do szyfrowania danych, stosując pogłębione analizy i inżynierię wsteczną.

W tym rozdziale przyjrzymy się bliżej ukrytym systemom plików i metodom ich analizowania. Podzielimy się naszymi doświadczeniami w wykony-

waniu długotrwałych analiz rootkitów i bootkitów opisanych w tej książce. Przedyskutujemy również podejścia do wydobywania danych z ukrytych magazynów i podzielimy się rozwiązaniami typowych problemów, które pojawiają się przy analizach tego rodzaju. Na koniec przedstawimy opracowane przez nas, niestandardowe narzędzie HiddenFsReader, którego celem jest zrzucenie zawartości ukrytych systemów plików tworzonych przez określone złośliwe programy.

Przegląd ukrytych systemów plików

Rysunek 18.1 ilustruje ogólny obraz typowego ukrytego systemu plików. Możemy tu zobaczyć złośliwy payload wstrzyknięty do przestrzeni adresowej trybu użytkownika procesu ofiary, który komunikuje się z ukrytą pamięcią masową. Payload często używa ukrytej pamięci masowej do wczytywania i aktualizowania swoich informacji konfiguracyjnych lub do przechowania danych, takich jak wykradzione poświadczenia.



Rysunek 18.1. Typowa implementacja złośliwego ukrytego systemu plików

Usługa ukrytej pamięci masowej udostępniana jest przez moduł trybu jądra, a interfejs eksponowany przez złośliwe oprogramowanie jest widoczny tylko dla modułu payloadu. Interfejs ten zazwyczaj nie jest dostępny dla innego oprogramowania w systemie i nie można do niego uzyskać dostępu przy użyciu metod standardowych, takich jak Eksplorator plików Windows.

Dane przechowywane przez złośliwe oprogramowanie w ukrytym systemie plików sa trwale przechowywane w obszarze dysku twardego, który nie jest używany przez system operacyjny, aby uniknąć konfliktu. W większości przypadków obszar ten znajduje się na końcu dysku twardego, gdyż zazwyczaj znajduje się tam nieco nieprzydzielonego miejsca. Jednak w niektórych przypadkach, takich jak bootkit Rovnix omówiony w rozdziale 11, złośliwe oprogramowanie może umieścić swój ukryty system plików w nieprzydzielonym miejscu na początku dysku twardego.

Głównym celem badacza wykonującego analizę jest wydobycie tych ukrytych danych, zatem omówimy teraz kilka podejść do realizacji tego celu.

Odczytywanie danych bootkitu z ukrytego systemu plików

Informacje można uzyskać z zainfekowanego komputera, odczytując dane przy wyłączonym zainfekowanym systemie (offline) albo przez czytanie złośliwych danych z aktywnego, zainfekowanego systemu.

Każde z tych podejść ma swoje zalety i wady, które uwzględnimy przy omawianiu obu metod.

Odczytywanie danych z systemu w stanie offline

Rozpoczniemy od uzyskiwania danych z dysku twardego, gdy system jest w stanie offline (czyli gdy złośliwe oprogramowanie nie jest aktywne). Można to osiągnąć przez analizę dysku twardego w trybie offline, ale alternatywnym wyborem jest uruchomienie niezainfekowanego wystąpienia systemu operacyjnego przy użyciu *live CD*. Gwarantuje to, że komputer użyje nieprzejętego programu ładującego zainstalowanego na CD, zatem bootkit nie zostanie uruchomiony. To podejście zakłada, że bootkit nie jest w stanie wykonać się przed uprawnionym programem ładującym i że nie będzie w stanie wykryć próby rozruchu z urządzenia zewnętrznego, aby zawczasu wymazać wrażliwe dane.

Znaczącą przewagą tej metody nad analizą w stanie online jest to, że nie musimy zwalczać mechanizmów samoobrony złośliwego oprogramowania, które chronią zawartość ukrytej pamięci masowej. Jak zobaczymy w dalszych podrozdziałach, ominięcie takiej ochrony nie jest zadaniem trywialnym i wymaga pewnej specjalistycznej wiedzy.

UWAGA *Gdy już uzyskamy dostęp do danych przechowywanych na dysku twardym, możemy kontynuować, zrzucając obraz złośliwego ukrytego systemu plików, a następnie go rozszyfrować i analizować. Różne typy złośliwego oprogramowania wymagają różnych podejść do deszyfrowania i parsowania ukrytych systemów plików, co omówimy w podrozdziale „Analizowanie obrazu ukrytego systemu plików" na stronie 403.*

Wadą tej metody jest natomiast to, że wymaga zarówno fizycznego dostępu do przejętego komputera, jak i technicznych umiejętności uruchomienia go z aktywnego CD i zrzucenia ukrytego systemu plików. Spełnienie obu tych wymagań może być problematyczne.

Jeśli analizowanie nieaktywnej maszyny nie jest możliwe, musimy przyjąć podejście aktywne.

Odczytywanie danych w działającym systemie

W działającym systemie z aktywnym wystąpieniem bootkitu potrzebujemy wykonać zrzut zawartości złośliwego, ukrytego systemu plików.

Odczytanie złośliwej ukrytej pamięci masowej w systemie z aktywnie działającym złośliwym oprogramowaniem wiąże się jednak z podstawową trudnością: oprogramowanie to może przeciwdziałać próbom odczytu i fałszować dane czytane z dysku twardego, aby wstrzymać analizę śledczą. Większość rootkitów omawianych w tej książce – TDL3, TDL4, Rovnix, Olmasco itd. – monitoruje dostęp do dysku twardego i blokuje próby dostępu do regionów zawierających złośliwe dane.

Aby móc odczytać podstępne dane z dysku twardego, musimy pokonać mechanizmy samoobrony złośliwego oprogramowania. Za chwilę przyjrzymy się kilku metodom wykonania tego zadania, ale najpierw zbadamy stos sterowników urządzeń pamięci masowych systemów Windows i to, jak złośliwe oprogramowanie zahacza się w nim, aby lepiej zrozumieć, jak próbuje ono chronić swoje dane. Informacja ta jest również przydatna do zrozumienia pewnych podejść do usuwania złośliwych hooków.

Zahaczanie sterownika miniportu urządzenia pamięci masowej

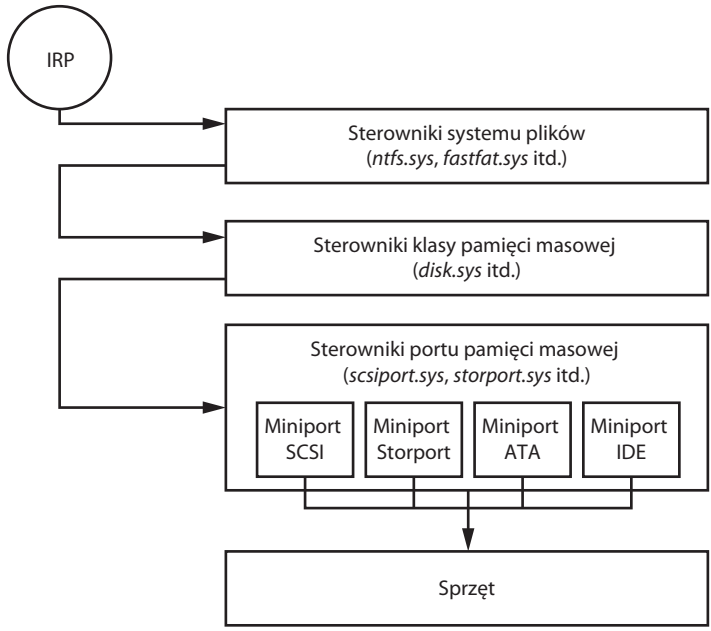
Z architekturą sterowników urządzeń pamięci masowej w systemach Microsoft Windows zetknęliśmy się już w rozdziale 1. Ta metoda zapewniała przetrwanie TDL3 i została zaadoptowana przez późniejsze złośliwe programy, w tym bootkity, które studiowaliśmy w tej książce. Teraz zajmiemy się dalszymi szczegółami.

TDL3 zahaczał sterownik miniportu urządzenia pamięci masowej zlokalizowany na samym spodzie stosu sterowników urządzeń, co jest pokazane na rysunku 18.2.

Zahaczenie w stosie sterowników na tym poziomie umożliwia złośliwemu oprogramowaniu monitorowanie i modyfikowanie żądania wejścia/wyjścia przekazywanego do i z dysku twardego, zapewniając mu dostęp do ukrytej pamięci masowej.

Zahaczenie na samym spodzie stosu sterowników i bezpośrednie komunikowanie się ze sprzętem pozwala również ominąć oprogramowanie zabezpieczające działające na poziomie systemu plików lub sterowniku klasy dysku. Jak wspomnieliśmy w rozdziale 1, gdy na dysku wykonywana jest operacja wejścia/wyjścia, system operacyjny generuje pakiet żądania wejścia/

wyjścia (*input/output request packet* – IRP) – specjalną strukturę danych w jądrze systemu operacyjnego opisującą operację wejścia/wyjścia – który jest przekazywany przez cały stos sterowników od szczytu do spodu.



Rysunek 18.2. Stos sterowników urządzeń pamięci masowej

Moduły oprogramowania zabezpieczającego odpowiedzialne za monitorowanie operacji wejścia/wyjścia dysku twardego mogą badać i modyfikować pakiety IRP, ale ponieważ złośliwe hooki są zainstalowane poniżej poziomu tego oprogramowania, są one niewidzialne dla tych narzędzi.

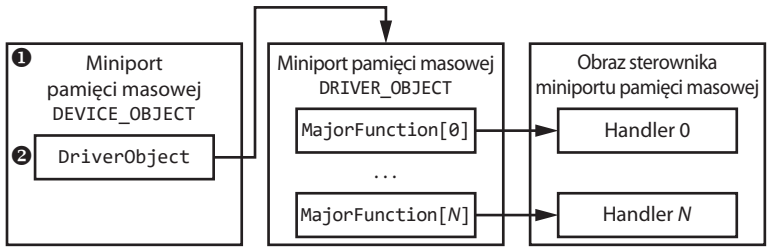
Istnieje też wiele innych poziomów, na których bootkit mógłby się zahaczyć, takich jak API trybu użytkownika, sterownik systemu plików lub sterownik klasy dysku, ale żaden z nich nie pozwoliłby złośliwemu oprogramowaniu na taką skrytość i siłę jak poziom miniportu.

Układ stosu urządzeń pamięci masowej

W tym punkcie nie będziemy omawiać wszystkich możliwych metod zahaczania miniportu pamięci masowej. Zamiast tego skupimy się na najbardziej typowych podejściach, z jakimi zetknęliśmy się w trakcie naszych analiz.

Na początek przyjrzyjmy się bliżej urządzeniu pamięci masowej pokazanemu na rysunku 18.3.

Pakiet IRP przechodzi od szczytu stosu do jego spodu. Każde urządzenie w stosie może albo przetworzyć i ukończyć żądanie wejścia/wyjścia, albo przekazać je do urządzenia jeden poziom niżej.



Rysunek 18.3. Organizacja miniportu urządzenia pamięci masowej

`DEVICE_OBJECT` ❶ jest systemową strukturą danych używaną przez system operacyjny do opisu urządzenia w stosie, zawierającą wskaźnik ❷ do odpowiadającej mu innej systemowej struktury danych, `DRIVER_OBJECT`, która opisuje sterownik załadowany w systemie. W tym przypadku `DEVICE_OBJECT` zawiera wskaźnik do sterownika miniportu pamięci masowej.

Układ struktury `DRIVER_OBJECT` pokazany jest w listingu 18.1.

```
typedef struct _DRIVER_OBJECT {
   SHORT Type;
   SHORT Size;
❶ PDEVICE_OBJECT DeviceObject;
   ULONG Flags;
❷ PVOID DriverStart;
❸ ULONG DriverSize;
   PVOID DriverSection;
   PDRIVER_EXTENSION DriverExtension;
❹ UNICODE_STRING DriverName;
   PUNICODE_STRING HardwareDatabase;
   PFAST_IO_DISPATCH FastIoDispatch;
❺ LONG * DriverInit;
   PVOID DriverStartIo;
   PVOID DriverUnload;
❻ LONG * MajorFunction[28];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

Listing 18.1. Układ struktury DRIVER_OBJECT

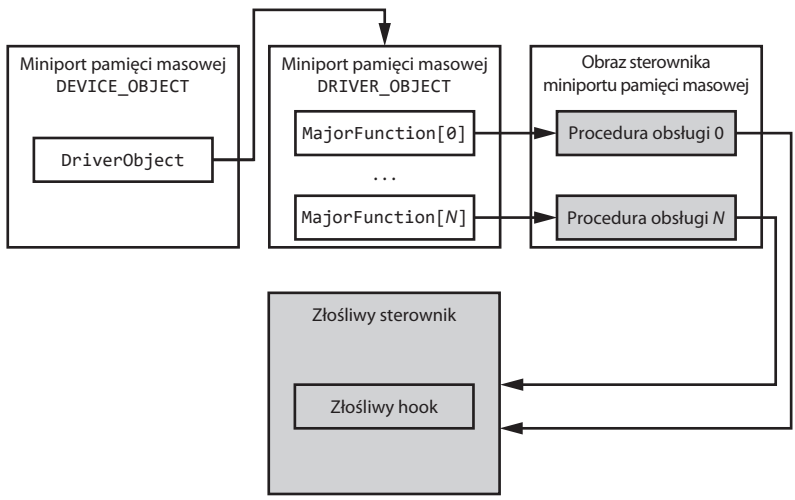
Kolejne pola zawierają: `DriverName` ❹ – nazwę sterownika opisywanego przez strukturę, `DriverStart` ❷ i `DriverSize` ❸ – odpowiedni adres początkowy oraz rozmiar w pamięci sterownika, `DriverInit` ❺ – wskaźnik do procedury inicjowania sterownika, a `DeviceObject` ❶ – wskaźnik do listy struktur `DEVICE_OBJECT` powiązanych ze sterownikiem. Z punktu widzenia złośliwego oprogramowania najważniejszym polem jest `MajorFunction` ❻,

zlokalizowane na końcu struktury, które zawiera adresy procedur obsługi implementowane w sterowniku dla różnych operacji wejścia/wyjścia.

Gdy pakiet wejścia/wyjścia dociera do obiektu urządzenia, system operacyjny sprawdza pole `DriverObject` w odpowiadającej mu strukturze `DEVICE_OBJECT`, aby uzyskać adres `DRIVER_OBJECT` w pamięci. Gdy jądro znajdzie strukturę `DRIVER_OBJECT`, wyszukuje adres odpowiedniej procedury obsługi wejścia/wyjścia z tablicy `MajorFunction`, odpowiednio do typu operacji. Mając te informacje, możemy zidentyfikować części stosu urządzenia pamięci masowej, które mogą zostać zahaczone przez złośliwe oprogramowanie. Przyjrzyjmy się kilku różnym metodom.

Bezpośrednie łatanie obrazu sterownika miniportu pamięci masowej

Jednym ze sposobów na zahaczenie sterownika miniportu jest bezpośrednia modyfikacja obrazu sterownika w pamięci. Gdy złośliwe oprogramowanie uzyska adres obiektu urządzenia miniportu dysku twardego, zagląda do `DriverObject`, aby zlokalizować odpowiadajacą mu strukturę `DRIVER_OBJECT`. Następnie wyszukuje adres procedury obsługi wejścia/wyjścia dysku w tablicy `MajorFunction` i łata kod pod tym adresem, co widać na rysunku 18.4 (części zamalowane na szaro są modyfikowane przez złośliwe oprogramowanie).



Rysunek 18.4. Zahaczanie stosu sterowników pamięci masowej przez łatanie sterownika miniportu

Gdy obiekt urządzenia otrzyma żądanie wejścia/wyjścia, wykonywane jest złośliwe oprogramowanie. Złośliwy hook może teraz odrzucić operacje wejścia/wyjścia, aby zablokować dostęp do chronionego obszaru dysku twardego, albo może zmodyfikować żądanie, aby zwrócić sfałszowane dane i oszukać oprogramowanie zabezpieczające.

Ten typ hooka był przykładowo używany przez bootkit Gapz omówiony w rozdziale 12. W przypadku Gapz złośliwe oprogramowanie zahaczało dwie procedury sterownika miniportu dysku twardego, które są odpowiedzialne za żądania `IRP_MJ_INTERNAL_DEVICE_CONTROL` oraz `IRP_MJ_DEVICE_CONTROL`, aby uchronić się przed odczytaniem lub nadpisaniem.

Niemniej jednak to podejście nie jest szczególnie skryte. Oprogramowanie zabezpieczające może wykryć i usunąć hooki, lokalizując obraz zahaczonego sterownika w systemie plików i mapując go do pamięci. Następnie porównuje sekcje kodu sterownika załadowanego do jądra z wersją sterownika ręcznie załadowanego z pliku i zauważa dowolne różnice w sekcjach kodu, które mogą sygnalizować obecność złośliwych hooków w sterowniku.

Oprogramowanie zabezpieczające może następnie usunąć złośliwe hooki i przywrócić oryginalny kod, nadpisując kod zmodyfikowany tym odczytanym z pliku. Metoda ta zakłada, że sterownik w systemie plików jest oryginalny i nie został zmodyfikowany przez złośliwe oprogramowanie.

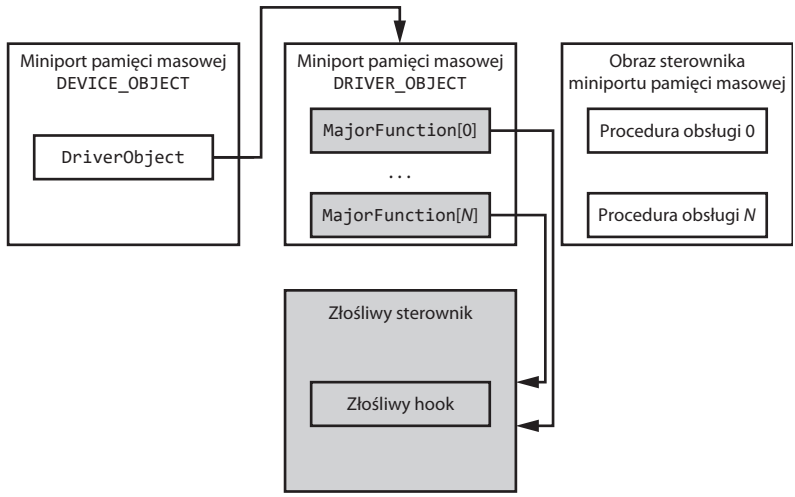
Modyfikacja DRIVER_OBJECT

Sterownik miniportu dysku twardego można również zahaczyć przez modyfikację struktury `DRIVER_OBJECT`. Jak wspomniano, ta struktura danych zawiera lokalizację obrazu sterownika w pamięci oraz adresy procedur udostępnianych przez sterownik w tablicy `MajorFunction`.

Tym samym zmodyfikowanie tablicy `MajorFunction` pozwala złośliwemu oprogramowaniu na zainstalowanie swoich hooków bez dotykania obrazu sterownika w pamięci. Zamiast łatania kodu bezpośrednio w obrazie, jak w poprzedniej metodzie, złośliwe oprogramowanie może na przykład zastąpić wpisy w tablicy `MajorFunction` odpowiadające żądaniom `IRP_MJ_INTERNAL_DEVICE_CONTROL` oraz `IRP_MJ_DEVICE_CONTROL` adresami złośliwych hooków. W rezultacie jądro systemu operacyjnego zostanie przekierowane do złośliwego kodu, ilekroć spróbuje odczytać adresy procedur obsługi ze struktury `DRIVER_OBJECT`. To podejście jest zademonstrowane na rysunku 18.5.

Ponieważ obraz sterownika pozostał niezmieniony, to podejście jest lepiej ukryte od poprzedniej metody, ale nadal nie jest niepodatne na próby odkrycia. Oprogramowanie zabezpieczające nadal może wykryć obecność hooków, lokalizując obraz sterownika w pamięci i sprawdzając adresy procedur obsługi żądań `IRP_MJ_INTERNAL_DEVICE_CONTROL` oraz `IRP_MJ_DEVICE_CONTROL`: jeśli te adresy nie należą do zakresu adresów obrazu sterownika miniportu w pamięci, wskazuje to, że w stosie urządzeń występują hooki.

Jednakże usunięcie tych hooków i przywrócenie oryginalnych wartości tablicy `MajorFunction` jest znacznie trudniejsze niż w poprzedniej metodzie. Przy tym podejściu tablica `MajorFunction` jest inicjowana przez sam sterownik podczas wykonywania jego procedury inicjującej, która otrzymuje wskaźnik do częściowo zainicjowanej struktury `DRIVER_OBJECT` jako parametr wejściowy i dopełnia inicjowanie, wypełniając tablicę `MajorFunction` wskaźnikami do procedur obsługi dyspozytorów.



Rysunek 18.5. Zahaczanie stosu sterowników pamięci masowej przez łatanie obiektu DRIVER_OBJECT miniportu

Tylko sterownik miniportu zna adresy procedur obsługi. Oprogramowanie zabezpieczające nie ma żadnej wiedzy na ich temat, co znacznie utrudnia przywrócenie oryginalnych adresów w strukturze `DRIVER_OBJECT`.

Jedno z możliwych podejść, którego może użyć oprogramowanie zabezpieczające w celu przywrócenia oryginalnych danych, polega na załadowaniu obrazu sterownika miniportu w emulowanym środowisku, utworzeniu struktury `DRIVER_OBJECT` i wykonaniu punktu wejściowego sterownika (procedury inicjowania) ze strukturą `DRIVER_OBJECT` jako parametrem. Po zakończeniu procedury inicjowania `DRIVER_OBJECT` powinna zawierać poprawne procedury obsługi w tablicy `MajorFunction` i można użyć tych informacji do obliczenia adresów procedur dyspozytorów wejścia/wyjścia w obrazie sterownika i przywrócić zmodyfikowaną strukturę `DRIVER_OBJECT`.

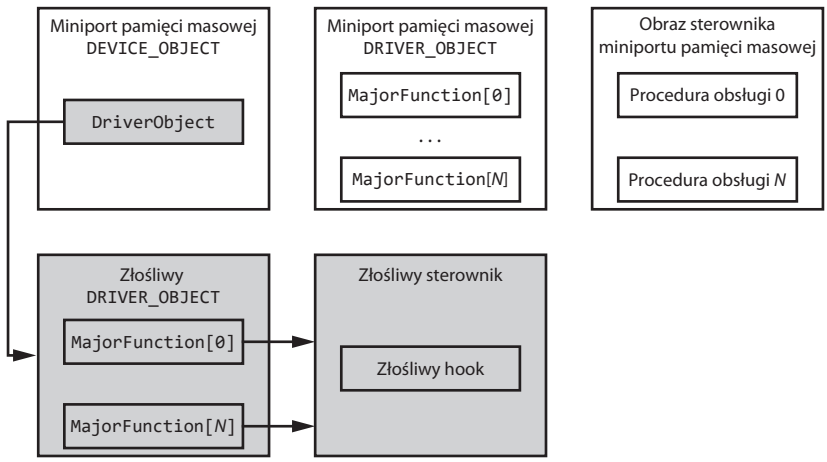
Emulowanie sterownika może jednak być niełatwe. Jeśli procedura inicjowania sterownika implementuje prostą funkcjonalność (np. zainicjowanie struktury `DRIVER_OBJECT` poprawnymi adresami procedur obsługi), to podejście powinno zadziałać, ale jeśli zawiera złożone działania (takie jak wywoływanie usług lub API systemu, które są trudniejsze do emulowania), emulacja może się nie udać i zakończyć, zanim sterownik zainicjuje strukturę danych. W takich przypadkach oprogramowanie zabezpieczające nie będzie w stanie przywrócić adresów oryginalnych procedur obsługi i usunąć złośliwych hooków.

Inne podejście do tego problemu to wygenerowanie bazy danych z oryginalnymi adresami i wykorzystanie jej do przywracania. Rozwiązaniu temu jednak brakuje ogólności. Może się sprawdzić dla najczęściej używanych sterowników miniportów, ale zawiedzie w przypadku rzadszych lub niestandardowych sterowników, które nie zostały uwzględnione w bazie danych.

Modyfikacja DEVICE_OBJECT

Ostatnie podejście do zahaczania sterownika miniportu, które uwzględnimy w tym rozdziale, jest logiczną kontynuacją poprzedniej metody. Wiemy, że aby wykonać procedurę obsługi żądania wejścia/wyjścia w sterowniku miniportu, jądro systemu operacyjnego musi wydobyć adres struktury `DRIVER_OBJECT` z `DEVICE_OBJECT` miniportu, następnie adres procedury obsługi z tablicy `MajorFunction`, a na koniec wykonać tę procedurę.

Zatem inną metodą zainstalowania hooka może być modyfikacja pola `DriverObject` w powiązanym `DEVICE_OBJECT`. Złośliwe oprogramowanie musi utworzyć oszukańczą strukturę `DRIVER_OBJECT` i zainicjować jej tablicę `MajorFunction` adresami złośliwych hooków, po czym jądro systemu użyje tej struktury `DRIVER_OBJECT` do uzyskiwania adresów procedur obsługi żądań wejścia/wyjścia i wykona złośliwy hook (rysunek 18.6).



Rysunek 18.6. Zahaczanie stosu sterowników pamięci masowej przez przejęcie DRIVER_OBJECT miniportu

To podejście jest wykorzystywane przez TDL3/TDL4, Rovnix i Olmasco; zapewnia podobne korzyści i ma podobne wady jak poprzednia metoda. Jednak takie hooki są jeszcze trudniejsze do usunięcia, gdyż cały `DRIVER_OBJECT` jest inny, co oznacza, że oprogramowanie zabezpieczające musi wykonać dodatkowe działania, aby zlokalizować oryginalną strukturę `DRIVER_OBJECT`.

To kończy nasze omówienie technik zahaczania stosu sterowników urządzeń. Jak pokazaliśmy, nie istnieje proste generyczne rozwiązanie usuwania złośliwych hooków, aby móc odczytać podstępne dane z chronionych obszarów dysku twardego zainfekowanej maszyny. Innym źródłem trudności jest to, że istnieje wiele różnych implementacji sterowników miniportu pamięci masowej, a ponieważ komunikują się one bezpośrednio ze sprzętem, każdy

dostawca urządzeń zapewnia własne sterowniki dla swojego sprzętu. Tym samym podejścia, które działają dobrze dla pewnych klas sterowników miniportów, zawiodą dla innych.

Analizowanie obrazu ukrytego systemu plików

Gdy już uda się zdeaktywować samoobronę rootkitu, możemy odczytać dane ze złośliwego ukrytego magazynu, co daje nam obraz złośliwego systemu plików. Kolejnym logicznym krokiem analizy jest badanie ukrytego systemu plików i wydobycie znaczących informacji.

Aby móc analizować zrzucony system plików, musimy znać typ złośliwego oprogramowania, któremu on odpowiada. Każde zagrożenie ma własną implementację ukrytej pamięci masowej, a jedyną metodą rekonstrukcji jej układu jest inżynieria wsteczna złośliwego oprogramowania, aby zrozumieć kod odpowiedzialny za jej utrzymanie. W niektórych przypadkach układ ukrytego systemu plików może zmieniać się między wersjami w ramach tej samej rodziny złośliwego oprogramowania.

Złośliwe oprogramowanie może również szyfrować lub zaciemniać swój ukryty system plików, aby utrudnić wykonywanie analizy, a w takim przypadku musimy dodatkowo znaleźć klucze szyfrujące.

Tabela 18.1 przedstawia podsumowanie ukrytych systemów plików powiązanych z rodzinami złośliwego oprogramowania, które omówiliśmy we wcześniejszych rozdziałach. W tej tabeli uwzględniamy tylko podstawowe cechy ukrytego systemu plików, takie jak typ układu, stosowane szyfrowanie i to, czy implementuje on kompresję.

Tabela 18.1. Porównanie implementacji ukrytych systemów plików

Funkcjonalność/złośliwe oprogramowanie	TDL4	Rovnix	Olmasco	Gapz
Typ systemu plików	własny	modyfikacja FAT16	własny	własny
Szyfrowanie	XOR/RC4	własne (XOR+ROL)	modyfikacja RC6	RC4
Kompresja	nie	tak	nie	tak

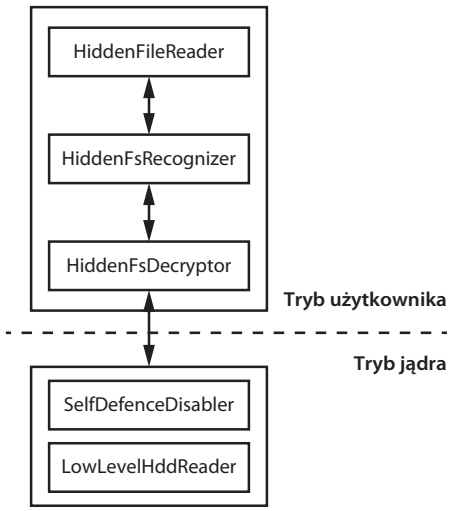
Jak widać, każda implementacja jest inna, co stwarza trudności analitykom śledczym i badaczom.

Narzędzie HiddenFsReader

W trakcie naszych badań nad zaawansowanymi zagrożeniami wykonaliśmy inżynierię wsteczną wielu różnych rodzin takich programów i zdołaliśmy zgromadzić bogatą bazę informacji o różnych implementacjach ukrytych

systemów plików, która może być przydatna społeczności badania bezpieczeństwa. Z tego powodu zbudowaliśmy narzędzie o nazwie HiddenFsReader (*http://download.eset.com/special/ESETHfsReader.exe/*), które automatycznie szuka ukrytych złośliwych kontenerów w komputerze i wydobywa zawarte w nich informacje.

Na rysunku 18.7 jest pokazany schemat architektury narzędzia HiddenFsReader.



Rysunek 18.7. Schemat architektury HiddenFsReader

Narzędzie HiddenFsReader składa się z dwóch komponentów: aplikacji trybu użytkownika oraz sterownika trybu jądra. Sterownik trybu jądra zasadniczo implementuje funkcjonalność blokowania mechanizmów samoobrony rootkitu/bootkitu, a aplikacja trybu użytkownika udostępnia interfejs umożliwiający uzyskanie niskopoziomowego dostępu do dysku twardego. Aplikacja wykorzystuje ten interfejs do odczytywania rzeczywistych danych z dysku twardego, nawet jeśli system jest zainfekowany aktywnym wystąpieniem złośliwego oprogramowania.

Sama aplikacja trybu użytkownika jest odpowiedzialna za zidentyfikowanie ukrytych systemów plików wczytywanych z dysku twardego, a dodatkowo implementuje funkcjonalność deszyfrowania, pozwalającą uzyskać jawny tekst z zaszyfrowanego ukrytego systemu plików.

W ostatnim wydaniu narzędzia HiddenFsReader (w chwili pisania tych słów) obsługiwane są następujące zagrożenia i odpowiadające im ukryte systemy plików:

- Win32/Olmarik (TDL3/TDL3+/TDL4),
- Win32/Olmasco (MaxXSS),

- Win32/Sirefef (ZeroAccess),
- Win32/Rovnix,
- Win32/Xpaj,
- Win32/Gapz,
- Win32/Flamer,
- Win32/Urelas (GBPBoot),
- Win32/Avatar.

Zagrożenia te stosują własne, ukryte systemy plików do przechowywania danych payloadu i konfiguracji, lepszej ochrony przed oprogramowaniem zabezpieczającym i utrudnienia analizy. Nie omówiliśmy w tej książce wszystkich tych zagrożeń, ale informacje o nich można znaleźć w liście źródeł dostępnej pod adresem *https://nostarch.com/rootkits/*.

Podsumowanie

Implementacja własnego, ukrytego systemu plików jest typowa w zaawansowanych zagrożeniach, takich jak rootkity i bootkity. Ukryta pamięć masowa służy do poufnego przechowywania informacji konfiguracyjnych i payloadów, sprawiając, że tradycyjne podejścia do badania są nieskuteczne.

Analityk musi wyłączyć mechanizmy samoobrony zagrożenia i wykonać inżynierię wsteczną złośliwego oprogramowania. W ten sposób może zrekonstruować układ ukrytego systemu plików i zidentyfikować schemat szyfrowania oraz klucz użyty do ochrony złośliwych danych. Wymaga to dodatkowego czasu i starań w stosunku do każdego zagrożenia, ale w tym rozdziale przedstawiliśmy kilka możliwych podejść do rozwiązywania tych problemów. W rozdziale 19 będziemy kontynuować poznawanie metod badania złośliwego oprogramowania, skupiając się na rootkitach UEFI. Przedstawimy informacje o uzyskiwaniu oprogramowania układowego UEFI i jego analizie z punktu widzenia złośliwego oprogramowania atakującego UEFI.

19

ŚLEDZTWA BIOS/UEFI: PODEJŚCIA DO ZDOBYWANIA OPROGRAMOWANIA UKŁADOWEGO I ANALIZ

Niedawne rootkity wymierzone w oprogramowanie układowe UEFI odświeżyły zainteresowanie badaniami bezpieczeństwa UEFI. Wycieki poufnych informacji o implantach BIOS-u tworzonych przez służby specjalne, a także wspomniane w rozdziale 15 włamanie do Hacking Team zademonstrowały coraz bardziej skryte i skuteczne możliwości złośliwego oprogramowania ukierunkowanego na BIOS i skłoniły społeczność badaczy do głębszych poszukiwań w oprogramowaniu układowym. Omówiliśmy już pewne szczegóły techniczne tych zagrożeń dla BIOS-u w poprzednich rozdziałach. Jeśli ktoś nie przeczytał rozdziałów 15 i 16, zdecydowanie zalecamy zrobić to teraz przed kontynuowaniem lektury; rozdziały te wyjaśniają ważne koncepcje zabezpieczeń oprogramowania układowego, które, jak zakładamy, Czytelnik już zna i rozumie.

UWAGA *W tym rozdziale wymiennie używamy terminów* BIOS *i* oprogramowanie układowe UEFI.

Badania śledcze oprogramowania układowego UEFI są obecnie rozwojowym obszarem badawczym, zatem specjaliści zabezpieczeń pracujący w tej dziedzinie odczuwają brak konwencjonalnych narzędzi i podejść. W tym rozdziale przedstawimy kilka technik analizy oprogramowania układowego, w tym różne podejścia do zdobywania oprogramowania układowego oraz metody jego parsowania i wydobywania przydatnych informacji.

Najpierw skupimy się na uzyskiwaniu oprogramowania układowego, co jest zazwyczaj pierwszym krokiem analizy. Przedstawimy zarówno programowe, jak i sprzętowe podejścia do uzyskiwania obrazu oprogramowania układowego UEFI, po czym porównamy te podejścia i przedyskutujemy ich zalety i wady. Następnie omówimy wewnętrzną strukturę oprogramowania układowego UEFI i sposoby jego analizy w celu wydobycia artefaktów dowodowych. W kontekście tego omówienia pokażemy, jak korzystać z UEFITool, nieodzownego narzędzia open source analizy oprogramowania układowego do przeglądania i modyfikowania obrazów UEFI. Na koniec omówimy Chipsec, narzędzie o bardzo szerokiej i efektywnej funkcjonalności oraz rozważymy jego zastosowania w analizach śledczych. Oba narzędzia przedstawiliśmy w rozdziale 15.

Ograniczenia naszych technik badania

Materiał przedstawiony tutaj ma pewne ograniczenia. W nowoczesnych platformach występuje wiele typów oprogramowania układowego: UEFI, Intel ME, oprogramowanie kontrolerów dysków twardych itd. Rozdział ten jest poświęcony głównie analizom oprogramowania układowego UEFI, które stanowi jedną z największych części oprogramowania platform.

Trzeba też zauważyć, że oprogramowanie układowe jest bardzo powiązane z platformą; innymi słowy, platforma ma własne osobliwości. W tym rozdziale skupimy się na oprogramowaniu układowym UEFI dla systemów Intel x86, które stanowią ogromną większość segmentów rynkowych komputerów biurkowych, laptopów i serwerów.

Dlaczego badanie oprogramowania układowego jest ważne

W rozdziale 15 widzieliśmy, że nowoczesne oprogramowanie układowe jest wygodnym miejscem wbudowania bardzo skutecznych backdoorów lub rootkitów, szczególnie wewnątrz BIOS-u. Ten typ złośliwego oprogramowania jest w stanie przetrwać ponowną instalację systemu operacyjnego lub wymianę dysku twardego i daje napastnikowi kontrolę nad całą platformą. W chwili pisania tych słów większość najbardziej zaawansowanego oprogra-

mowania zabezpieczającego w ogóle nie uwzględniało zagrożeń dotyczących oprogramowania układowego UEFI, przez co stają się one jeszcze groźniejsze. Daje to napastnikowi ogromną okazję implantowania złośliwego oprogramowania, które pozostanie niewykryte w atakowanym systemie.

Następnie naszkicujemy kilka konkretnych sposobów, na jakie napastnicy mogliby wykorzystać rootkity oprogramowania układowego.

Atakowanie łańcucha dostaw

Zagrożenia wymierzone w oprogramowanie układowe UEFI zwiększają ryzyko ataków na łańcuch dostaw, gdyż napastnicy mogą zainstalować złośliwy implant w serwerze, zanim zostanie on dostarczony do centrum danych, albo w laptopie, zanim trafi on w ręce działu IT przedsiębiorstwa. A ponieważ zagrożenia te mogą wpływać na wielką liczbę klientów dostawcy usługi przez ujawnienie ich wszystkich sekretów, wielcy gracze w obszarze przetwarzania chmurowego, tacy jak Google, zaczęli ostatnio używać technik badania oprogramowania układowego w celu upewnienia się, że używane przez nich oprogramowanie nie zostało przejęte.

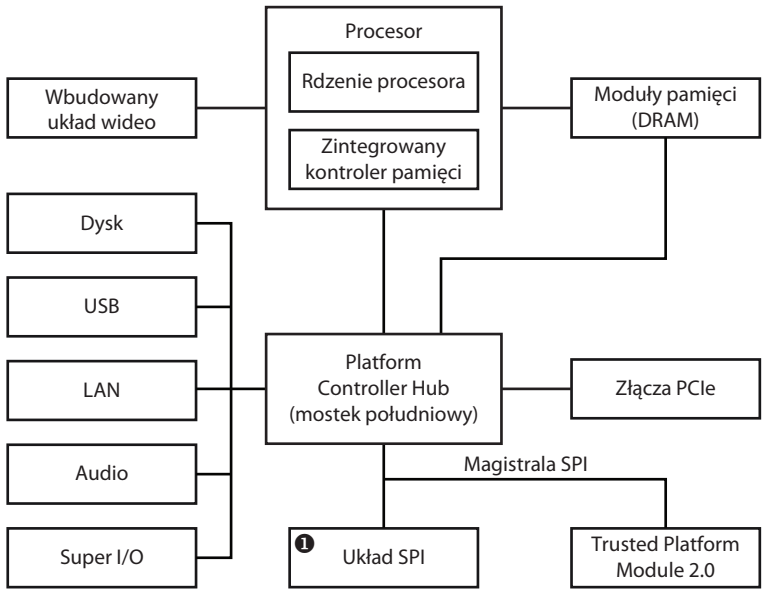
> **Chip Titan firmy Google**
>
> Firma Google w 2017 roku publicznie przedstawiła Titan, układ chroniący oprogramowanie układowe platformy przez ustanowienie sprzętowego źródła zaufania. Zaufanie do konfiguracji sprzętowej jest ważną rzeczą, szczególnie w przypadku zabezpieczeń chmury, gdzie wpływ ataku jest zwielokrotniony liczbą dotkniętych nim klientów.
>
> Firmy pracujące z wielkimi chmurami i danymi, takie jak Amazon, Google, Microsoft, Facebook i Apple, pracują nad opracowaniem (albo już opracowały) sprzętu kontrolującego źródło zaufania platformy. Nawet jeśli napastnicy użyją rootkitu oprogramowania układowego do przejęcia platformy, posiadanie odizolowanego źródła zaufania powstrzyma ataki na Secure Boot i na aktualizacje oprogramowania układowego.

Przełamanie zabezpieczeń BIOS-u przez podatność oprogramowania układowego

Napastnicy mogą przejąć oprogramowanie układowe platformy, wykorzystując istniejącą w nim podatność do ominięcia ochrony zapisu BIOS-u lub uwierzytelniania. Aby odświeżyć sobie informacje na temat takiego ataku, warto powrócić do rozdziału 16, w którym omawiamy różne klasy podatności używane do ataków na BIOS. W celu wykrycia takich ataków można użyć omówionych w tym rozdziale podejść do analizy śledczej oprogramowania układowego w celu weryfikacji integralności oprogramowania platformy albo jako pomoc w wykrywaniu złośliwych modułów oprogramowania układowego.

Pozyskiwanie oprogramowania układowego

Pierwszym krokiem analizy śledczej dotyczącej BIOS-u jest proces uzyskania obrazu oprogramowania układowego BIOS do tej analizy. Aby lepiej zrozumieć lokalizację oprogramowania BIOS w nowoczesnych platformach, będziemy odwoływać się do rysunku 19.1, na którym jest pokazany schemat architektury chipsetu typowego systemu komputera osobistego.



Rysunek 19.1. Schemat blokowy nowoczesnego chipsetu Intel

W chipsecie występują dwa główne komponenty: procesor (CPU) oraz Platform Controller Hub (PCH), nazywany też *mostkiem południowym* (South Bridge). PCH zapewnia połączenie między kontrolerami urządzeń peryferyjnych dostępnych w platformie a procesorem. W większości nowoczesnych systemów opartych na architekturze Intel x86 (włącznie z platformami 64-bitowymi) systemowe oprogramowanie układowe jest zlokalizowane w układzie pamięci flash na magistrali Serial Peripheral Interface (SPI) ❶, która jest fizycznie połączona z PCH. Układ flash SPI stanowi główny cel analizy śledczej, gdyż przechowuje oprogramowanie układowe, które chcemy analizować.

Płyta główna komputera PC zawiera typowo jeden, wlutowany w nią, oddzielny fizyczny układ flash SPI, ale można niekiedy spotkać systemy zawierające wiele układów flash SPI. Dzieje się tak, gdy pojedynczy układ nie ma dostatecznej pojemności, aby przechować całe oprogramowanie układowe systemu; w takim przypadku dostawca platformy może użyć dwóch układów. Sytuację taką omówimy w podrozdziale „Lokalizowanie układu pamięci flash SPI” na stronie 421.

Technologia DualBIOS

Technologia DualBIOS również wykorzystuje wiele układów flash SPI na płycie głównej komputera. Jednak w odróżnieniu od właśnie omówionego podejścia, w którym wiele układów flash SPI przechowuje jeden obraz oprogramowania układowego, DualBIOS używa wielu układów do przechowywania różnych obrazów oprogramowania lub wielokrotności tego samego obrazu. Ta technologia zapewnia dodatkową ochronę przed uszkodzeniem oprogramowania układowego, gdyż jeśli oprogramowanie w jednym układzie będzie uszkodzone, system może wykonać rozruch z drugiego układu zawierającego identyczny obraz oprogramowania.

W celu uzyskania obrazu oprogramowania układowego przechowanego w układzie flash SPI musimy być w stanie odczytać zawartość pamięci flash. Mówiąc ogólnie, można odczytać oprogramowanie układowe przy użyciu podejścia programowego lub sprzętowego. W podejściu programowym próbujemy odczytać obraz oprogramowania układowego, komunikując się z kontrolerem SPI przy użyciu oprogramowania uruchomionego na procesorze hosta. W podejściu sprzętowym fizycznie dołączamy do układu flash SPI specjalne urządzenie nazywane programatorem SPI, po czym odczytujemy obraz oprogramowania układowego bezpośrednio z układu flash SPI. Przedstawimy tu oba podejścia, zaczynając od metody programowej.

Zanim jednak przejdziemy do opisu podejścia programowego, trzeba wiedzieć, że każde z tych podejść ma swoje zalety i ograniczenia. Jedną z zalet zrzucania oprogramowania układowego UEFI metodą programową jest to, że można to zrobić zdalnie. Użytkownik docelowego systemu może uruchomić aplikację zrzucającą zawartość układu flash SPI, po czym wysłać ją do analizy. Jednak podejście to ma również wielką wadę: jeśli napastnik zdołał już przejąć oprogramowanie układowe systemu, może zakłócić proces wydobywania oprogramowania, fałszując dane odczytywane z układu flash SPI. Sprawia to, że podejście programowe może okazać się niepewne.

Podejście sprzętowe jest wolne od tej wady. Wprawdzie musimy być fizycznie obecni i musimy otworzyć obudowę systemu, jednak tą metodą bezpośrednio wczytujemy zawartość wyłączonej pamięci flash SPI, nie dając napastnikowi żadnej szansy podrabiania danych (chyba że mamy do czynienia z implantem sprzętowym, którym to zagadnieniem nie zajmujemy się w tej książce).

Podejście programowe do uzyskiwania oprogramowania układowego

W podejściu programowym do zrzucenia oprogramowania układowego UEFI z badanego systemu odczytujemy zawartość pamięci flash SPI z poziomu systemu operacyjnego. Dostęp do kontrolerów SPI nowoczesnych systemów można uzyskać przez rejestry w obszarze konfiguracji PCI (blok

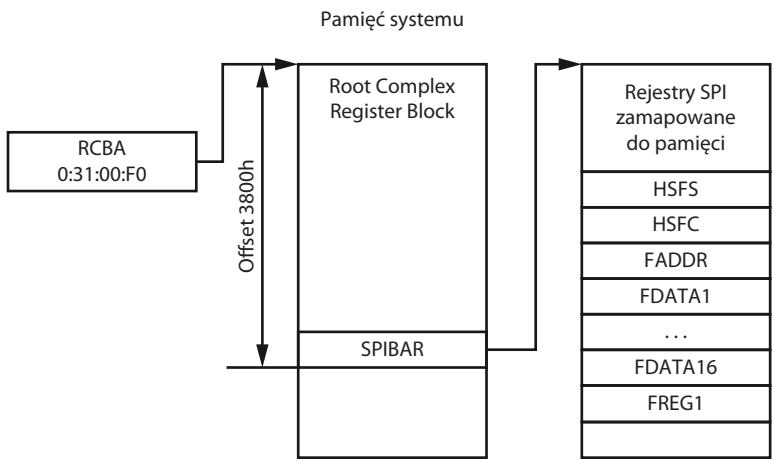
rejestrów specyfikujących konfigurację urządzeń na magistrali PCI). Rejestry te są mapowane do pamięci i można je odczytywać i zapisywać przy użyciu zwykłych operacji odczytu i zapisu pamięci. W tym podrozdziale pokażemy, jak zlokalizować te rejestry i komunikować się z kontrolerem SPI.

Zanim przejdziemy dalej, trzeba wiedzieć, że lokalizacja rejestru SPI jest specyficzna dla chipsetu, zatem aby móc komunikować się z kontrolerem SPI, musimy odwołać się do chipsetu dedykowanego dla badanej platformy. W tym rozdziale zademonstrujemy, jak odczytać pamięć flash SPI w chipsetach występujących w Intel 200 Series (lokalizację rejestrów SPI można znaleźć na stronie *https://www.intel.com/content/www/us/en/chipsets/200-series-chipset-pch-datasheet-vol-2.html*), które w chwili pisania tych słów stanowiły najnowsze chipsety dla systemów biurkowych.

Warto także zauważyć, że lokalizacje pamięci odpowiadające rejestrom eksponowanych za pośrednictwem obszaru konfiguracji PCI są mapowane na przestrzeń adresową trybu jądra, a w rezultacie nie są dostępne dla kodu działającego w przestrzeni adresowej trybu użytkownika. Aby uzyskać dostęp do tego zakresu adresów, konieczne jest przygotowanie sterownika trybu jądra. Narzędzie Chipsec omówione w dalszej części tego rozdziału udostępnia własny sterownik trybu jądra w celu uzyskania dostępu do przestrzeni konfiguracji PCI.

Lokalizowanie rejestrów obszaru konfiguracji PCI

Najpierw musimy zlokalizować zakres pamięci, do którego mapowane są rejestry kontrolera SPI. Ten zakres pamięci nosi nazwę *Root Complex Register Block* (RCRB). Wewnątrz RCRB pod offsetem 3800h znajdziemy rejestr *SPI Base Address Register* (SPIBAR), który zawiera adres bazowy zamapowanych do pamięci rejestrów SPI (rysunek 19.2).



Rysunek 19.2. Lokalizacja rejestrów sterowania i statusu SPI w pamięci systemu

Magistrala PCIe

Magistrala PCI Express (PCIe) jest standardem magistrali szeregowej dużych prędkości, używanym w praktycznie wszystkich współczesnych komputerach osobistych, we wszystkich segmentach rynku: konsumenckich laptopów i komputerów biurkowych, serwerów itd. Magistrala PCIe służy jako łącznik między różnymi komponentami i urządzeniami peryferyjnymi komputera. Wiele urządzeń zintegrowanych w chipsecie (układy flash SPI, kontroler pamięci itd.) jest reprezentowanych jako punkty końcowe PCIe na magistrali.

Adres RCRB jest przechowywany w rejestrze PCI *Root Complex Base Address (RCBA)* zlokalizowanym na magistrali (*bus*) 0, urządzeniu (*device*) 31h, funkcji (*function*) 0. Jest to rejestr 32-bitowy, adres RCRB zawarty jest zaś w bitach 31:14. Przyjmujemy, że dolne 14 bitów adresu RCRB to zera, gdyż RCRB jest wyrównany do granicy 16 Kb. Po znalezieniu adresu RCRB możemy uzyskać wartość SPIBAR, odczytując pamięć pod offsetem 3800h. W kolejnym podrozdziale bardziej szczegółowo omówimy rejestry SPI.

Oprogramowanie układowe we flash SPI

Układ flash SPI zawiera nie tylko oprogramowanie układowe BIOS, lecz także inne typy oprogramowania platformy, takie jak Intel ME (*Manageability Engine*), oprogramowanie kontrolera Ethernet oraz oprogramowanie i dane specyficzne dla określonego dostawcy. Te różne typy oprogramowania różnią się lokalizacją i uprawnieniami kontroli dostępu. Na przykład, system operacyjny hosta nie może uzyskać dostępu do oprogramowania Intel ME, zatem podejście programowe do uzyskiwania oprogramowania układowego nie sprawdzi się w przypadku Intel ME.

Obliczanie adresów rejestru konfiguracji SPI

Po uzyskaniu wartości SPIBAR informującej o położeniu rejestrów SPI w pamięci możemy zaprogramować rejestry, aby odczytać zawartość flash SPI. Offsety rejestrów SPI mogą zmieniać się w zależności od platformy, zatem najlepszym sposobem ustalenia rzeczywistych wartości dla danej konfiguracji sprzętowej jest wyszukanie wartości w dokumentacji chipsetu platformy. Na przykład dla platform obsługujących najnowsze (w chwili pisania tych słów) procesory Intela (Kaby Lake) można odwołać się do karty danych Intel 200 Series Chipset Family Platform Controller Hub, aby znaleźć lokalizację rejestrów SPI mapowanych do pamięci. Informacja ta znajduje się w sekcji „Serial Peripheral Interface". Dla każdego rejestru SPI karta danych podaje jego offset względem wartości SPIBAR, nazwę rejestru oraz wartość domyślną po zresetowaniu platformy. W tym podrozdziale użyjemy tej karty danych jako źródła w celu ustalenia adresów interesujących nas rejestrów.

Korzystanie z rejestrów SPI

Teraz, gdy już wiemy, jak znaleźć adresy rejestrów SPI, można ustalić, którego z nich możemy użyć, aby odczytać zawartość pamięci flash SPI. W tabeli 19.1 są podane wszystkie rejestry, których potrzebujemy, aby uzyskać obraz pamięci flash SPI.

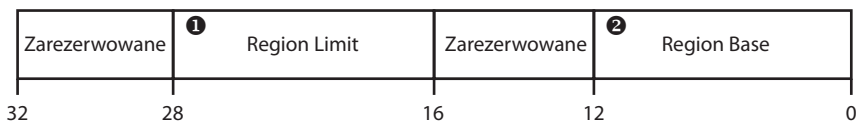
Tabela 19.1. Rejestry SPI potrzebne dla pobierania oprogramowania układowego

Offset względem SPIBAR	Nazwa rejestru	Opis rejestru
04h–05h	HSFS	status sekwencjonowania sprzętowego pamięci flash
06h–07h	HSFC	rejestr kontroli sekwencjonowania sprzętowego pamięci flash
08h–0Bh	FADDR	adres pamięci flash
10h–4Fh	FDATAX	tablica danych pamięci flash
58h–5Bh	FREG1	region 1 pamięci flash (deskryptor BIOS-u)

Każdy z tych rejestrów omówimy w kolejnych punktach.

Rejestr FREG1

Rejestr, od którego zaczniemy, to *region 1 pamięci flash* (*FREG1*). Udostępnia on lokalizację regionu BIOS w pamięci flash SPI. Układ tego rejestru o długości 32 bitów jest przedstawiony na rysunku 19.3.

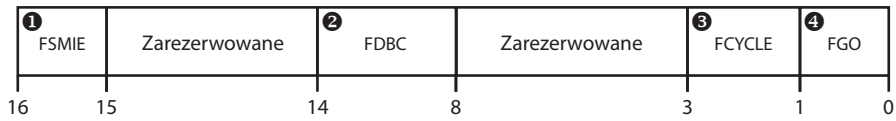


Rysunek 19.3. Układ rejestru FREG1 SPI

Pole Region Base ❷ udostępnia bity 24:12 adresu bazowego regionu BIOS w pamięci flash SPI. Ponieważ region BIOS-u jest wyrównany do 4 Kb, dolne 12 bitów adresu bazowego regionu zaczyna się od 0. Pole Region Limit ❶ zawiera 24:12 bitów dla regionu BIOS-u w pamięci SPI. Jeśli na przykład pole Region Base zawiera wartość 0xaaa, a Region Limit wartość 0xbbb, to region BIOS-u w pamięci flash SPI rozciąga się od 0xaaa000 do 0xbbbfff.

Rejestr HSFC

Rejestr *kontroli sekwencjonowania sprzętowego pamięci flash* (*hardware sequencing flash control* – HSFC) umożliwia wysyłanie poleceń do kontrolera SPI. (W specyfikacji polecenia te są określane mianem *cykli*). Układ rejestru HSFC jest pokazany na rysunku 19.4.



Rysunek 19.4. Układ rejestru HSFC SPI

Rejestru HSFC używamy do wysyłania cykli odczytu/zapisu/usuwania do pamięci flash SPI. 2-bitowe pole FCYCLE ❸ koduje operację do wykonania następująco:

00 – odczytanie bloku danych z pamięci flash SPI;

01 – zapisanie bloku danych w pamięci flash SPI;

11 – wymazanie bloku danych z pamięci flash SPI;

10 – zarezerwowane.

W przypadku cykli odczytu i zapisu pole FDBC ❷ wskazuje liczbę bajtów, która mają zostać przesłane do lub z flash SPI. Zawartość tego pola jest liczona od zera; wartość 000000b oznacza 1 bajt, a wartość 111111b – 64 bajty. Innymi słowy, liczba bajtów do przesłania to wartość w tym polu plus 1.

Pole FGO ❹ służy do inicjowania operacji flash SPI. Jeśli wartość w tym polu to 1b, kontroler SPI będzie odczytywać, zapisywać i wymazywać dane, zgodnie z polami FCYCLE i FDBC. Przed ustawieniem pola FGO oprogramowanie musi wyspecyfikować wszystkie pola, wskazujące typ operacji, ilość danych oraz adres w pamięci flash SPI.

Ostatnim polem rejestru HSFC, które zasługuje na uwagę, jest pole *flash SPI SMI# enable* (FSMIE) ❶. Gdy to pole jest ustawione, chipset generuje przerwanie zarządzania systemem (*System Management Interrupt* – SMI), co powoduje wykonanie kodu SMM. Jak zobaczymy w podrozdziale „Uwzględnianie niedoskonałości podejścia programowego" na stronie 417, można użyć FSMIE, aby przeciwdziałać uzyskiwaniu obrazu oprogramowania układowego.

Komunikowanie się z kontrolerem SPI

Wykorzystanie rejestru HSFC nie jest jedynym sposobem przesyłania poleceń do kontrolera SPI. W ogólności istnieją dwie metody komunikowania się z układem flash SPI: sekwencjonowanie sprzętowe lub programowe. W pokazanej tu metodzie sekwencjonowania sprzętowego pozwalamy sprzętowi wybrać polecenia SPI, które zostana wysłane w celu wykonania operacji odczytu/zapisu (właśnie do tego celu służy rejestr HSFC). Sekwencjonowanie programowe zapewnia większe możliwości wyboru określonych poleceń, które zostaną przesłane w celu wykonania operacji odczytu/zapisu. W tym punkcie użyliśmy sekwencjonowania sprzętowego za pośrednictwem rejestru HSFC, gdyż jest łatwe i zapewnia funkcjonalność potrzebną do odczytania oprogramowania układowego BIOS.

Rejestr FADDR

Rejestr *flash address* (FADDR) pozwala określić adres liniowy w pamięci flash SPI operacji odczytu, zapisu i wymazywania. Rejestr ten jest 32-bitowy, ale do wskazywania liniowego adresu operacji są używane tylko dolne 24 bity. Górnych 8 bitów tego rejestru jest zarezerwowane i nieużywane.

Rejestr HSFS

Po zainicjowaniu cyklu SPI przez ustawienie pola FGO rejestru HSFC można ustalić, czy cykl się zakończył, sprawdzając rejestr *statusu sekwencjonowania sprzętowego pamięci flash* (*hardware sequencing flash status* – HSFS). Rejestr ten składa się z kilku pól, które zwracają informacje o statusie żądanej operacji. W tabeli 19.2 można zobaczyć pola HSFS używane do odczytywania obrazu SPI.

Tabela 19.2. Pola rejestru HSFS

Offset pola	Rozmiar pola	Nazwa pola	Opis pola
0h	1	FDONE	cykl flash zakończony
1h	1	FCERR	błąd cyklu flash
2h	1	AEL	wpis błędu dostępu
5h	1	SCIP	cykl SPI w trakcie wykonywania

Bit FDONE jest ustawiony przez chipset, gdy poprzedni cykl flash (zainicjowany przez pole FGO rejestru HSFC) został ukończony. Bity FCERR oraz AEL odpowiednio sygnalizują wyst ąpienie błędu w trakcie cyklu flash SPI oraz to, że zwrócone dane mogą nie zawierać poprawnych wartości. Bit SCIP pokazuje, że cykl flash właśnie się odbywa. SCIP ustawiany jest przez ustawienie bitu FGO, a czyszczony – gdy wartość FDONE wynosi 1. Opierając się na tych informacjach, możemy ustalić, że zainicjowana przez nas operacja zakończyła się sukcesem, jeśli następujące wyrażenie jest prawdziwe:

```
(FDONE == 1) && (FCERR == 0) && (AEL == 0) && (SCIP == 0)
```

Rejestry FDATAX

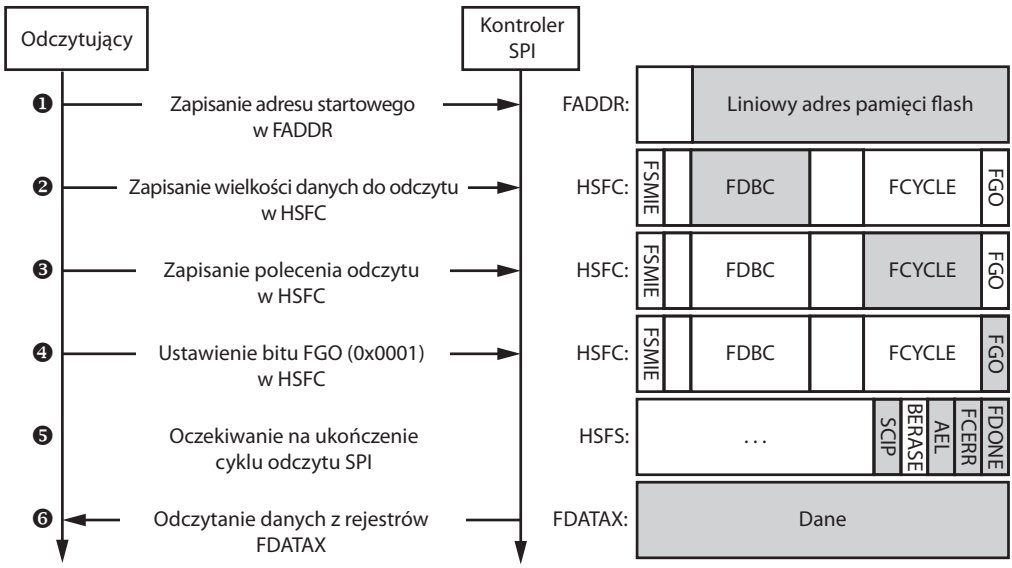
Rejestry tablic danych flash (*FDATAX*) przechowują dane, które mają być zapisane lub odczytane z pamięci flash SPI. Każdy rejestr to 32 bity, a całkowita liczba rejestrów FDATAX zależy od liczby bajtów do przesłania, specyfikowanych w polu FDBC rejestru HSFC.

Czytanie danych z pamięci flash SPI

Zbierzmy teraz razem wszystkie te informacje i zobaczmy, jak odczytać dane z pamięci flash SPI przy użyciu tych rejestrów. Najpierw lokalizujemy Root Complex Registers Block, na którego podstawie możemy ustalić

adres bazowy rejestrów SPI zamapowanych w pamięci i uzyskać do nich dostęp. Odczytując rejestr SPI FREG1, możemy ustalić lokalizację regionu BIOS-u w pamięci flash – czyli adres początkowy BIOS-u oraz limit.

Następnie możemy odczytać region BIOS-u przy użyciu opisanych przed chwilą rejestrów SPI. Krok ten jest pokazany na rysunku 19.5.



Rysunek 19.5. Odczytywanie danych z pamięci flash SPI

Najpierw ustawiamy rejestr FADDR na liniowy adres regionu pamięci flash, który chcemy odczytać ❶. Następnie określamy całkowitą liczbę bajtów do odczytania z pamięci flash, ustawiając pole FDBC ❷ rejestru kontroli. (Wartość 111111b spowoduje przeczytanie po 64 bajty w cyklu). Teraz ustawiamy w polu FCYCLE ❸ wartość 00b, co oznacza cykl odczytu, po czym ustawiamy bit FGO ❹, rozpoczynając operację odczytu.

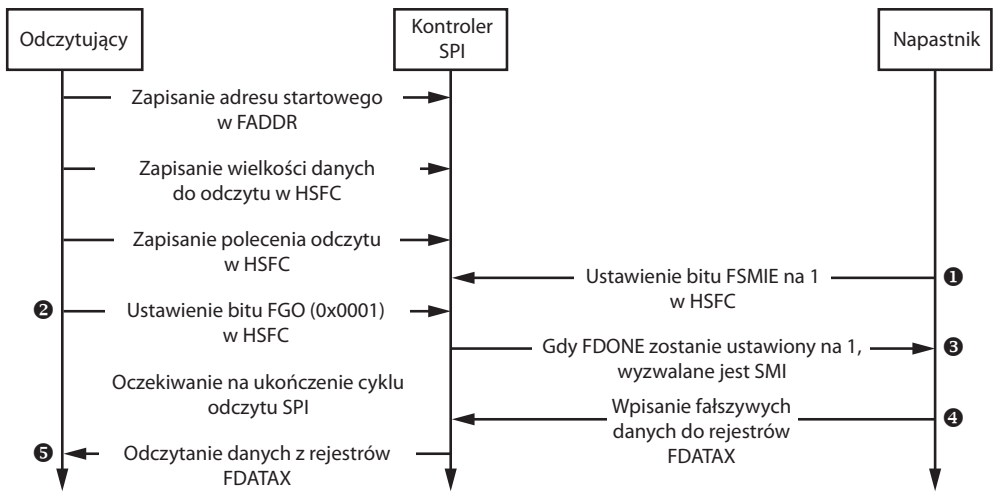
Po ustawieniu bitu FGO trzeba monitorować rejestr statusu, aby dowiedzieć się, kiedy operacja będzie ukończona. Można to zrobić, sprawdzając pola FDONE, FCERR, AEL i SCIP ❺. Po ukończeniu operacji odczytu wczytujemy dane flash z rejestrów FDATAX ❻. Rejestr FDATAX[1] udostępnia pierwsze 4 bajty pamięci flash pod adresem docelowym wskazywanym przez rejestr FADDR; FDATAX[2] zawiera kolejne 4 bajty itd. Powtarzając te kroki i zwiększając wartość FADDR o 64 bajty w każdej iteracji, możemy odczytać cały region BIOS-u z pamięci flash SPI.

Uwzględnianie niedoskonałości podejścia programowego

Programistyczne podejście do zrzucania oprogramowania układowego BIOS-u jest wygodne, gdyż nie wymaga fizycznej obecności; używając tej

metody można zdalnie odczytać zawartości pamięci flash SPI. Nie jest jednak odporne na napastnika, który już przejął systemowe oprogramowanie układowe i może wykonywać złośliwy kod w trybie SMM.

Jak wspomnieliśmy, rejestr HSFC zawiera bit FSMIE, który wyzwala SMI, gdy zakończy się cykl flash. Jeśli napastnik już przejął SMM i jest w stanie ustawić bit FSMIE, zanim program pobierający oprogramowanie układowe ustawiające bit FGO, to kod napastnika uzyska kontrolę po wygenerowaniu SMI i będzie w stanie zmodyfikować zawartość rejestrów FDATAX. W rezultacie program pobierający oprogramowanie układowe będzie czytać z FDATAX sfałszowane wartości i nie będzie w stanie uzyskać oryginalnego obrazu regionu BIOS-u. Atak taki demonstruje rysunek 19.6.



Rysunek 19.6. Zniekształcanie programowego pobierania BIOS-u za pomocą SMI

Zanim odczytujący ustawi bit FGO ❷ w rejestrze kontroli flash, napastnik wpisuje 1 do bitu FSMIE rejestru ❶. Gdy cykl się zakończy i dane zostaną zapisane z powrotem w rejestrach FDATAX, wyzwalane jest SMI i napastnik przejmuje sterowanie ❸. Następnie atakujący modyfikuje zawartość rejestrów FDATAX ❹, aby zamaskować atak na oprogramowanie układowe BIOS. Po odzyskaniu sterowania odczytujący otrzyma fałszywe dane ❺ i nie wykryje przejęcia oprogramowania układowego.

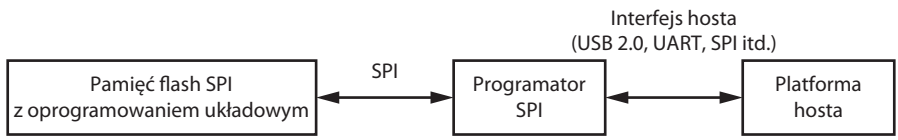
Atak ten demonstruje, że podejście programowe nie zapewnia w pełni niezawodnego podejścia do wydobywania oprogramowania układowego. W kolejnym podrozdziale omówimy podejście sprzętowe do tego zadania. Wykonywanie analizy przez fizyczne dołączenie urządzenia do pamięci flash SPI pozwala uniknąć możliwości ataku przedstawionego na rysunku 19.6.

Sprzętowe podejście do uzyskiwania oprogramowania układowego

Aby zagwarantować, że uzyskamy rzeczywisty obraz BIOS-u przechowywany w pamięci flash SPI, a nie sfałszowany przez napastnika, możemy użyć podejścia sprzętowego. W tej metodzie fizycznie podłączamy urządzenie do pamięci flash SPI i bezpośrednio odczytujemy jej zawartość. Jest to najlepszy sposób, gdyż jest bardziej godny zaufania od podejścia programowego. Dodatkową korzyścią jest to, że w ten sposób możemy uzyskać inne oprogramowanie układowe przechowywane w pamięci flash SPI, takie jak oprogramowanie ME i GBE, które może nie być dostępne w podejściu programowym ze względu na ograniczenia narzucane przez kontroler SPI.

Magistrala SPI w nowoczesnych systemach pozwala wielu urządzeniom nadrzędnym komunikować się z pamięcią flash SPI. Na przykład w systemach opartych na chipsetach Intela znajdziemy w ogólności trzy takie nadrzędne byty: procesor hosta, Intel ME oraz GBE. Mają one różne uprawnienia dostępu do różnych regionów pamięci flash SPI. W najnowszych platformach procesor hosta nie może odczytywać ani zapisywać regionu flash SPI zawierającego oprogramowanie układowe Intel ME i GBE.

Na rysunku 19.7 jest przedstawiona typowa konfiguracja stosowana przy uzyskiwaniu obrazu oprogramowania układowego BIOS przez odczytanie pamięci flash SPI.



Rysunek 19.7. Typowa konfiguracja dla zrzucania obrazu pamięci flash SPI

Aby móc odczytać dane z pamięci flash, potrzebujemy dodatkowego urządzenia, nazywanego *programatorem SPI*, które fizycznie podłączamy do układu pamięci flash SPI w docelowym systemie. Programator SPI łączymy przez interfejs USB lub UART do hosta, którego użyjemy do uzyskania obrazu oprogramowania BIOS. Następnie możemy uruchomić w programatorze pewne szczególne oprogramowanie, aby odczytać dane z układu pamięci flash i przesłać je do komputera analityka. Może być to własne oprogramowanie dołączone do konkretnego programatora SPI albo rozwiązanie open source, takie jak narzędzie Flashrom, które omówimy dalej w podrozdziale „Odczytywanie pamięci flash SPI przy użyciu minimodułu FT2232” na stronie 422.

Przegląd analizy przypadku Lenovo ThinkPad T540p

Podejście sprzętowe jest jeszcze bardziej swoiste od podejścia programowego. Wymaga sprawdzenia dokumentacji platformy, aby się dowiedzieć, jakiego rodzaju pamięć flash jest używana do przechowywania oprogramowania układowego i gdzie jest ona fizycznie zlokalizowana w systemie. Dodatkowo istnieje wiele różnych urządzeń programowania pamięci flash dla określonego sprzętu, których możemy użyć do odczytywania zawartości pamięci. Nie będziemy tu omawiać wszystkich sprzętowych i programowych możliwości uzyskiwania oprogramowania układowego, gdyż jest ich po prostu zbyt wiele. Zamiast tego przedstawimy jeden z możliwych sposobów zrzucenia oprogramowania układowego komputera Lenovo ThinkPad T540p z wykorzystaniem programatora SPI FT2232.

Wybraliśmy ten programator ze względu na jego stosunkowo niską cenę (około 30 USD) i elastyczność, a także na nasze wcześniejsze doświadczenia w korzystaniu z niego. Jak wspomnieliśmy, dostępnych jest wiele rozwiązań i każde ma swoje unikatowe funkcje, zalety i wady.

Programator Dediprog SF100 ISP IC

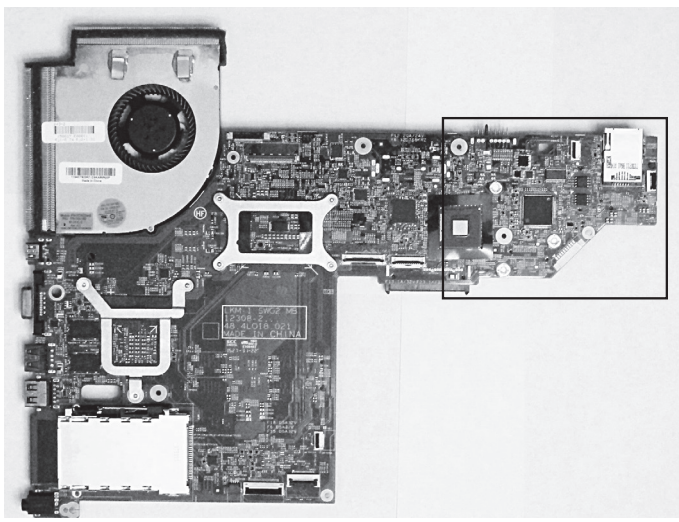
Innym urządzeniem, o którym chcielibyśmy wspomnieć, jest Dediprog SF100 ISP IC Programmer (pokazany na rysunku 19.8). Jest popularny w społeczności badaczy bezpieczeństwa sprzętu, obsługuje wiele rodzajów pamięci flash SPI i udostępnia rozległą funkcjonalność. Minnowboard, witryna źródeł open source dla projektantów sprzętu i oprogramowania układowego, zawiera dobry podręcznik wykorzystania Dediprog do aktualizowania oprogramowania pod adresem *https://minnowboard.org/tutorials/updating-firmware-via-spi-flash-programmer/*.

Rysunek 19.8. Programator Dediprog SF100 ISP IC

Lokalizowanie układu pamięci flash SPI

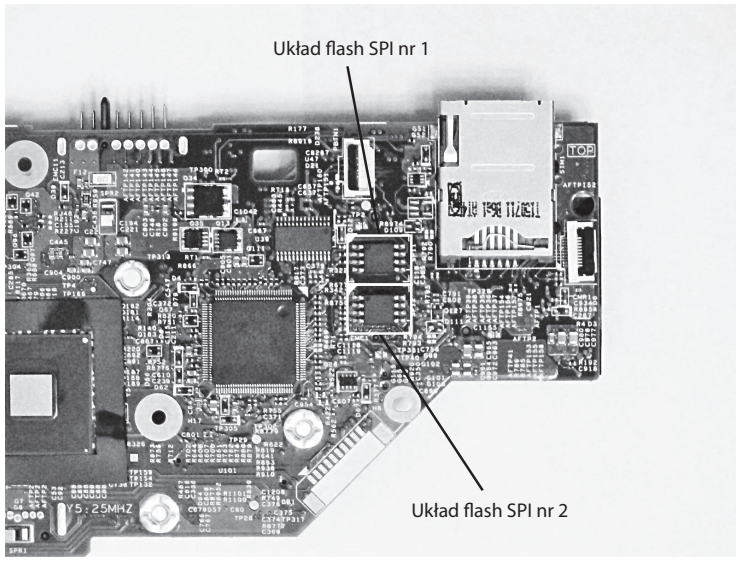
Zacznijmy od fizycznego odczytania obrazu oprogramowania układowego z platformy Lenovo ThinkPad T540p. Na początek, aby móc zrzucić systemowe oprogramowanie układowe z badanego systemu, musimy ustalić, gdzie na płycie głównej znajdują się układy pamięci flash SPI. W tym celu wykorzystaliśmy Hardware Maintenance Manual (*https://thinkpads.com/support/hmm/hmm_pdf/t540p_w540_hmm_en_sp40a26003_01.pdf*) tego modelu laptopa i rozebraliśmy komputer na części. Na rysunkach 19.9 i 19.10 można zobaczyć lokalizację dwóch układów pamięci flash. Na rysunku 19.9 jest pokazana cała systemowa płyta główna. Układy flash SPI są zlokalizowane w wyróżnionym obszarze.

UWAGA *Nie należy powtarzać działań opisanych w tym podrozdziale, jeśli nie ma się 100-procentowej pewności co do tego, co się robi. Nieodpowiednia lub niewłaściwa konfiguracja narzędzi może zniszczyć badany system.*



Rysunek 19.9. Płyta główna komputera Lenovo ThinkPad T540p z modułami flash SPI

Rysunek 19.10 jest zbliżeniem regionu wyróżnionego na rysunku 19.9, dzięki czemu lepiej widać układy flash SPI. W tym modelu laptopa są dwa moduły pamięci flash SOIC-8 służące do przechowywania oprogramowania układowego – jeden o pojemności 64 Mb (8 MB), a drugi o pojemności 32 Mb (4 MB). Jest to bardzo popularne rozwiązanie w wielu nowoczesnych komputerach biurkowych i laptopach.



Rysunek 19.10. Lokalizacja modułów pamięci flash SPI na płycie głównej laptopa

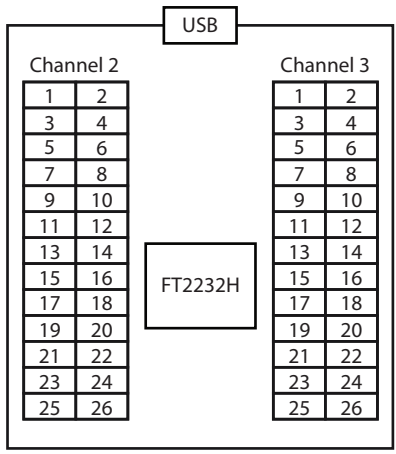
Ponieważ do przechowania systemowego oprogramowania układowego są użyte dwa układy, będziemy musieli zrzucić zawartość obu. Następnie uzyskamy finalny obraz oprogramowania układowego, łącząc obrazy tych układów pamięci w jeden plik.

Odczytywanie pamięci flash SPI przy użyciu modułu FT2232

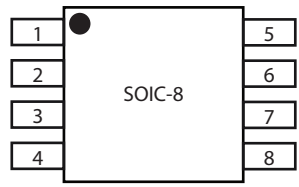
Po zidentyfikowaniu fizycznej lokalizacji układów możemy podłączyć styki programatora SPI do modułu flash na płycie głównej. Na karcie danych modułu FT2232H (*http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_FT2232H_Mini_Module.pdf*) widać, których styków należy użyć, aby podłączyć urządzenie do układu pamięci. Na rysunku 19.11 jest pokazany układ styków modułu FT2232H i układu flash SPI.

Urządzenie FT2232H zawiera dwa zestawy styków, odpowiadające dwóm kanałom: Channel 2 i Channel 3. Do odczytania zawartości pamięci flash SPI można użyć dowolnego z tych kanałów. W naszym eksperymencie użyliśmy Channel 3, aby podłączyć FT2232H do układu pamięci SPI. Na rysunku 19.11 widać, które styki FT2232H połączyliśmy z odpowiednimi wyprowadzeniami układu pamięci flash SPI.

Po podłączeniu urządzenia FT2232H do układu pamięci musimy je skonfigurować, aby działało w trybie zasilania z magistrali USB. Moduł FT2232H obsługuje dwa tryby działania: *zasilanie z magistrali USB* lub na *własnym zasilaniu*. W trybie zasilania z magistrali urządzenie pobiera prąd z podłączonej do niego magistrali USB, a w trybie własnego zasilania prąd dostarczany jest niezależnie od połączenia USB.



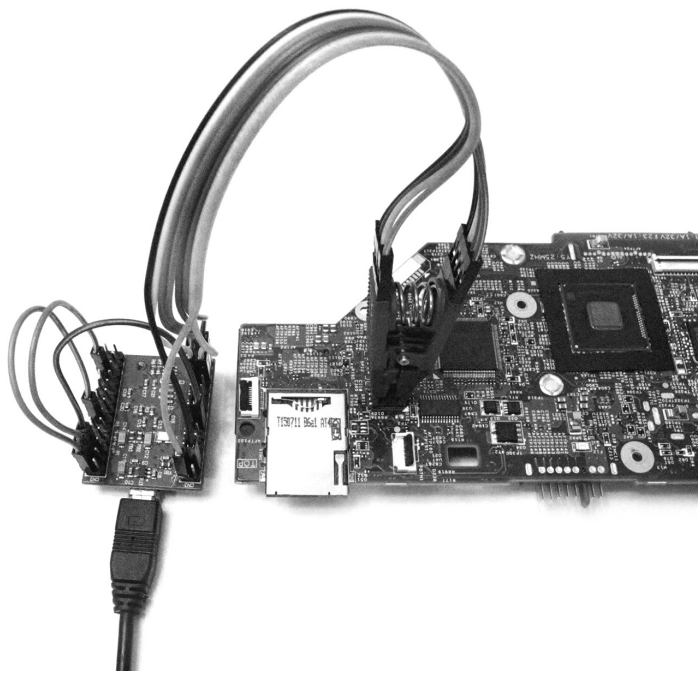
1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16
17	18
19	20
21	22
23	24
25	26



Styki Channel 3	Wyprowadzenia układu flash SPI
23	1
24	2
21	3
04	4
25	5
26	6
20	7
01	8

Rysunek 19.11. Układ styków minimodułu FT2232H oraz układu flash SPI

Jako pomoc w podłączeniu programatora do modułu SPI wykorzystaliśmy klip SOIC-8 pokazany na rysunku 19.12. Ten klip pozwala łatwo podłączyć złączki minimodułu z odpowiadającymi złączkami układu pamięci flash.



Rysunek 19.12. Podłączanie modułu FT2232H do układu pamięci flash SPI

Po podłączeniu wszystkich komponentów możemy odczytać zawartość układu flash SPI. W tym celu użyliśmy narzędzia open source o nazwie *Flashrom* (*https://www.flashrom.org/Flashrom*). Narzędzie to zostało opracowane specjalnie w celu identyfikowania, odczytywania, zapisywania, weryfikowania i wymazywania układów flash. Obsługuje wielką liczbę typów układów flash i działa z wieloma różnymi programatorami SPI, w tym z modułem FT2232H.

Listing 19.1 pokazuje wyniki uruchomienia Flashrom w celu odczytania obu układów flash SPI w platformie Lenovo ThinkPad T540p.

```
❶ user@host: flashrom -p ft2232_spi:type=2232H,port=B --read dump_1.bin
  flashrom v0.9.9-r1955 on Linux 4.8.0-36-generic (x86_64)
  flashrom is free software, get the source code at https://flashrom.org

  Calibrating delay loop... OK.
❷ Found Macronix flash chip "MX25L6436E/MX25L6445E/MX25L6465E/MX25L6473E"
  (8192 kB, SPI) on ft2232_spi.
❸ Reading flash... done.

  user@host: flashrom -p ft2232_spi:type=2232H,port=B --read dump_2.bin
  flashrom v0.9.9-r1955 on Linux 4.8.0-36-generic (x86_64)
  flashrom is free software, get the source code at https://flashrom.org

  Calibrating delay loop... OK.
  Found Macronix flash chip "MX25L3273E" (4096 kB, SPI) on ft2232_spi.
  Reading flash... done.

❹ user@host: cat dump_2.bin >> dump_1.bin
```

Listing 19.1. Zrzucanie obrazów flash SPI przy użyciu narzędzia Flashrom

Najpierw uruchomiliśmy Flashrom w celu zrzucenia zawartości pierwszego układu flash SPI, przekazując do niego jako parametry ❶ typ programatora i numer portu. Wyspecyfikowany typ 2232H odpowiada naszemu modułowi FT2232H, a portowi B odpowiada Channel 3, którego użyliśmy do połączenia się z układem flash SPI. Parametr `--read` nakazuje Flashrom odczytanie zawartości pamięci flash SPI i zapisanie jej do pliku *dump_1.bin*. Po uruchomieniu narzędzia wyświetlił on typ wykrytego układu flash SPI – w tym przypadku Macronix MX25L6473E ❷. Po zakończeniu odczytywania pamięci flash Flashrom wyświetla potwierdzenie ❸.

Po wczytaniu zawartości pierwszego układu flash przełączyliśmy klip do drugiego układu i ponownie uruchomiliśmy Flashrom, aby zrzucić zawartość drugiego układu do pliku *dump_2.bin*. Po wykonaniu tej operacji utworzyliśmy pełny obraz oprogramowania układowego, łącząc dwa zrzucone obrazy ❹.

Teraz mamy już pełny, godny zaufania obraz oprogramowania układowego. Nawet jeśli BIOS jest już zainfekowany i napastnik próbował udaremnić zdobycie tego oprogramowania, nadal jesteśmy w stanie uzyskać rzeczywisty kod i dane oprogramowania układowego. Teraz zajmijmy się jego analizą.

Analizowanie obrazu oprogramowania układowego przy użyciu UEFITool

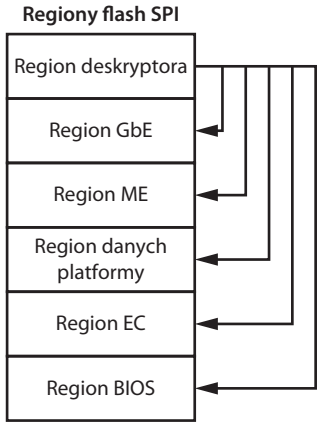
Po uzyskaniu obrazu oprogramowania układowego z pamięci flash SPI badanego systemu możemy go analizować. W tym podrozdziale zajmiemy się podstawowymi komponentami oprogramowania układowego platformy, takimi jak woluminy, pliki woluminów oraz sekcje, które są niezbędne do zrozumienia układu oprogramowania UEFI w obrazie pamięci flash. Następnie skupimy się na najważniejszych krokach analizy śledczej oprogramowania układowego.

UWAGA *W tym podrozdziale przedstawimy raczej omówienie wysokiego poziomu, a nie szczegółowe definicje używanych struktur danych, gdyż jest to zbyt rozległy temat i jego pogłębione omówienie wykracza poza zakres tego rozdziału. Przedstawimy jednak odsyłacze do dokumentacji zawierającej definicje i układ struktur danych dla tych Czytelników, którzy chcą uzyskać więcej informacji.*

Ponownie sięgniemy tu po narzędzie UEFITool (*https://github.com/LongSoft/UEFITool/*), narzędzie open source do analizy, wydobywania i modyfikowania obrazu oprogramowania układowego UEFI, które przedstawiliśmy w rozdziale 15, aby zademonstrować zastosowanie teoretycznych koncepcji wobec rzeczywistego obrazu oprogramowania układowego uzyskanego w poprzednim podrozdziale. W analizie śledczej możliwość obejrzenia obrazu oprogramowania układowego w celu przeglądania i wydobywania różnych komponentów jest niebywale użyteczna. Narzędzie to nie wymaga instalacji; po jego pobraniu aplikacja jest gotowa do użycia.

Poznawanie regionów pamięci flash SPI

Zanim spojrzymy na obraz oprogramowania układowego, musimy zająć się tym, jak uporządkowane są informacje przechowywane w pamięci flash SPI. W ogólności układy flash SPI nowoczesnych platform opartych na chipsecie Intel zawierają wiele obszarów. Każdy z nich jest przeznaczony do przechowania oprogramowania konkretnego urządzenia dostępnego w platformie; na przykład oprogramowanie układowe UEFI BIOS, Intel ME oraz Intel GBE (zintegrowanego urządzenia LAN) są przechowywane każde we własnym regionie. Na rysunku 19.13 jest pokazany układ kilku regionów pamięci flash SPI.



Rysunek 19.13. Regiony obrazu pamięci flash SPI

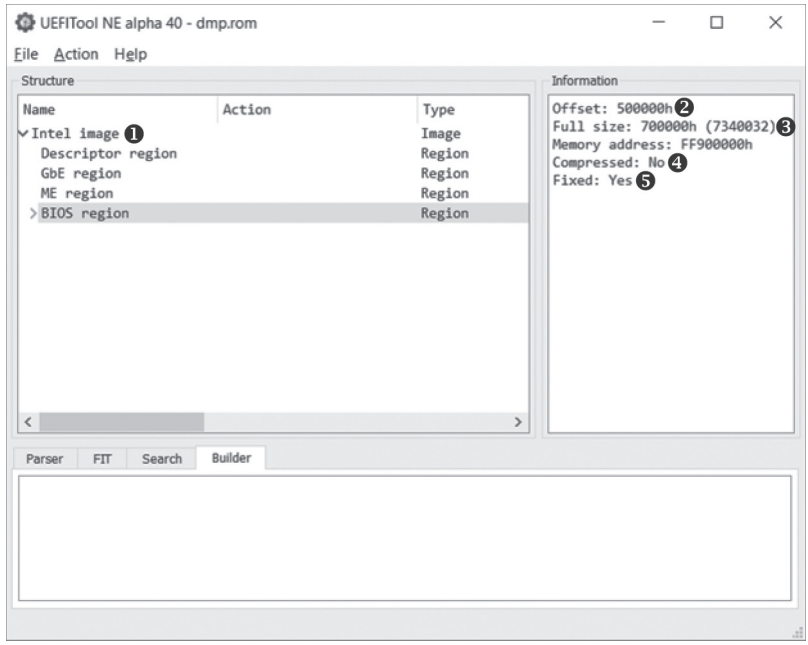
Pamięć flash SPI w nowoczesnych systemach obsługuje do sześciu regionów, w tym region deskryptora, od którego zawsze zaczynają się obrazy flash. Region deskryptora zawiera informacje o układzie pamięci flash SPI; innymi słowy, udostępnia chipsetowi informacje o innych regionach obecnych w pamięci flash SPI, takie jak ich lokalizacje i prawa dostępu. Region deskryptora określa również prawa dostępu każdego nadrzędnego bytu w systemie, który może komunikować się z kontrolerem flash SPI. Wiele obiektów nadrzędnych może się komunikować z kontrolerem w tym samym czasie. Pełny układ regionu deskryptora, w tym definicje wszystkich zawartych w nim struktur danych, możemy znaleźć w specyfikacji chipsetu docelowej platformy.

W tym rozdziale interesuje nas głównie region BIOS-u zawierający oprogramowanie układowe wykonywane przez procesor po wektorze resetu. Możemy wydobyć lokalizację regionu BIOS-u z regionu deskryptora. Zazwyczaj BIOS jest ostatnim regionem w pamięci flash SPI i stanowi ona główny cel analizy śledczej.

Przyjrzyjmy się różnym regionom obrazu SPI, który uzyskaliśmy przy użyciu podejścia sprzętowego.

Przeglądanie regionów pamięci flash SPI przy użyciu UEFITool

Najpierw uruchamiamy UEFITool i wybieramy polecenie **File ▸ Open image file**. Następnie wskazujemy plik zawierający obraz SPI do analizy – dostarczyliśmy taki plik w źródłach do tej książki pod adresem *https://nostarch.com/rootkits/*. Na rysunku 19.14 jest pokazany rezultat tej operacji.



Rysunek 19.14. Przeglądanie regionów pamięci flash SPI w UEFITool

Po załadowaniu obrazu oprogramowania układowego UEFITool parsuje go automatycznie i prezentuje informacje w postaci struktury drzewa. Jak widać na rysunku 19.14, narzędzie zidentyfikowało, że obraz oprogramowania układowego pochodzi z systemu opartego na chipsecie Intel ❶ i zawiera tylko cztery regiony SPI: deskryptor, ME, GbE oraz BIOS. Jeśli zaznaczymy region BIOS w oknie Structure, informacje na jego temat możemy zobaczyć w oknie Information. UEFITool pokazuje następujące elementy opisujące ten region:

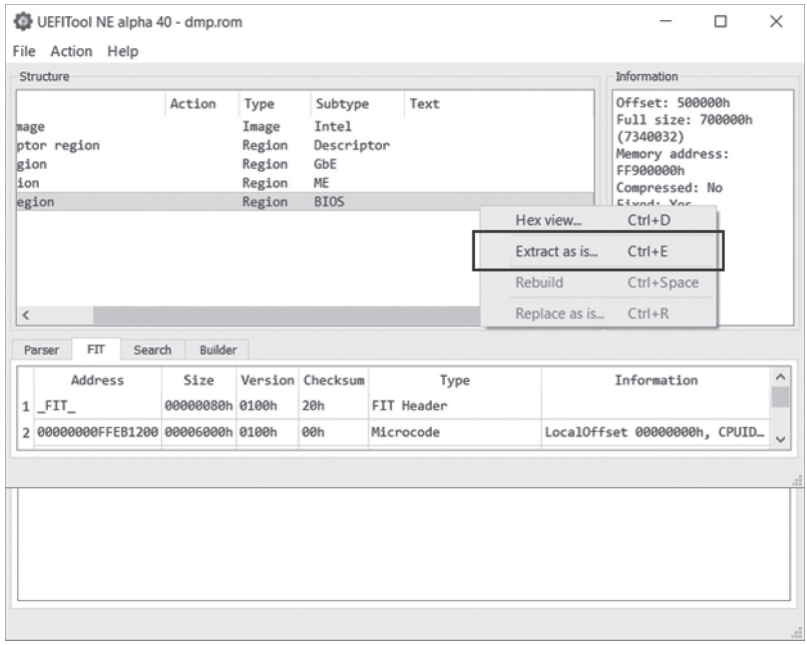
Offset ❷ – offset regionu względem początku obrazu flash SPI;

Full size ❸ – rozmiar regionu w bajtach;

Memory address ❹ – adres regionu po mapowaniu do pamięci fizycznej;

Compressed ❺ – informuje, czy region zawiera dane skompresowane.

Narzędzie udostępnia wygodną metodę wydobywania indywidualnych regionów (i dowolnych innych obiektów pokazywanych w oknie struktury) z obrazu SPI i zapisywania ich w oddzielnym pliku, co widać na rysunku 19.15.



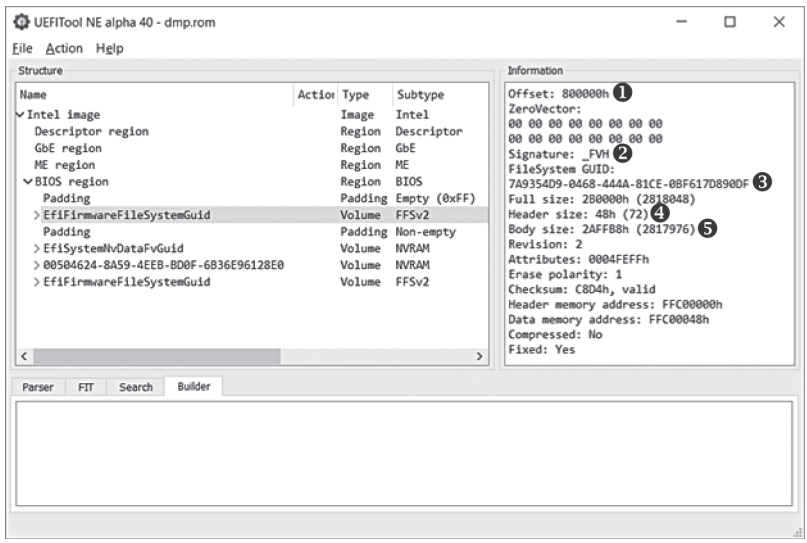
Rysunek 19.15. Wydobywanie regionu BIOS i zapisywanie go jako oddzielnego pliku

Aby wydobyć i zapisać region, należy kliknąć prawym klawiszem myszy wybrany region i wybrać polecenie **Extract as is ...** z menu kontekstowego. Narzędzie wyświetli następnie typowe okno dialogowe, w którym można wybrać nazwę i miejsce zapisania nowego pliku. Po wykonaniu tego polecenia warto sprawdzić wybraną lokalizację, aby potwierdzić, że operacja została zakończona z powodzeniem.

Analizowanie regionu BIOS-u

Po zidentyfikowaniu lokalizacji regionu BIOS-u możemy kontynuować analizę. Na wysokim poziomie region BIOS-u jest zorganizowany w woluminy oprogramowania układowego, które są podstawowymi repozytoriami magazynowymi danych i kodu. Dokładną definicję woluminu oprogramowania układowego można znaleźć w *EFI Firmware Volume Specification* (*https://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-firmware-file-volume-specification.html*). Każdy wolumin rozpoczyna się nagłówkiem zawierającym wymagane atrybuty woluminu, takie jak typ systemu plików woluminu, jego rozmiar oraz suma kontrolna.

Zbadajmy woluminy oprogramowania układowego dostępne w uzyskanym obrazie BIOS-u. Po dwukrotnym kliknięciu regionu BIOS-u w oknie narzędzia UEFITool (tak jak na rysunku 19.15) otrzymamy listę dostępnych woluminów, widoczną na rysunku 19.16.



Rysunek 19.16. Przeglądanie woluminów oprogramowania układowego dostępnych w regionie BIOS-u

W naszym regionie BIOS-u dostępne są cztery woluminy oprogramowania układowego oraz dwa regiony oznakowane jako *Padding* (dopełnienie). Regiony dopełnienia nie należą do żadnego woluminu oprogramowania układowego, ale oznaczają puste miejsce między nimi, wypełnione albo wartościami 0x00, albo 0xff, zależnie od polaryzacji wymazywania układu flash SPI. Polaryzacja wymazywania określa wartości zapisywane do pamięci flash podczas operacji wymazywania. Jeśli polaryzacja wymazywania to 1, to wymazane bajty pamięci flash otrzymują wartość 0xff; jeśli polaryzacja to 0, wymazywane bajty są ustawiane na 0x00. W rezultacie przy polaryzacji wymazywania równej 1 region dopełnienia (puste miejsce) składa się z wartości 0xff.

Na karcie informacyjnej widocznej na rysunku 19.16 na prawo od woluminów możemy zobaczyć atrybuty wybranego woluminu. Oto kilka istotnych pól:

Offset ❶ – offset woluminu oprogramowania układowego względem początku obrazu SPI;

Signature ❷ – sygnatura woluminu oprogramowania układowego w nagłówku; pole to służy do identyfikowania woluminów w regionach BIOS-u;

Filesystem GUID ❸ – identyfikator systemu plików używanego w woluminie oprogramowania układowego; ten globalnie unikatowy identyfikator (GUID) jest wyświetlany jako nazwa woluminu w oknie struktury; jeśli GUID jest udokumentowany, UEFITool wyświetla nazwę czytelną dla człowieka (taką jak EfiFirmwareFileSystemGuid, widoczną na rysunku 19.16), zamiast wartości heksadecymalnej;

Header size ❹ – rozmiar nagłówka woluminu oprogramowania układowego; dane woluminu następują po nagłówku;

Body size ❺ – rozmiar ciała woluminu oprogramowania układowego – czyli rozmiar danych przechowywanych w woluminie.

Poznawanie systemu plików oprogramowania układowego

Woluminy oprogramowania układowego są uporządkowane jako system plików, którego typ jest wskazywany przez GUID w nagłówku woluminu. Najczęściej używanym systemem plików w woluminach oprogramowania układowego jest *firmware filesystem* (*FFS*), zdefiniowany w specyfikacji EFI FFS, ale woluminy mogą również używać innych systemów plików, takich jak FAT32 lub NTFS. Skupimy się tu na FFS jako najbardziej popularnym.

FFS przechowuje wszystkie pliki w katalogu głównym i nie umożliwia tworzenia żadnej hierarchii katalogów. Zgodnie ze specyfikacją EFI FFS każdy plik ma przypisany typ, zlokalizowany w nagłówku tego pliku, opisujący dane przechowywane w tym pliku. Oto lista niektórych często spotykanych typów plików, które mogą być użyteczne w analizach śledczych:

EFI_FV_FILETYPE_RAW – surowy plik – nie należy przyjmować żadnych założeń na temat danych przechowywanych w tym pliku;

EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE – plik zawierający opakowany wolumin oprogramowania układowego; choć FFS nie udostępnia tworzenia hierarchii katalogów, można użyć tego typu pliku do utworzenia podobnej do drzewa struktury, opakowując moduły oprogramowania układowego w pliki;

EFI_FV_FILETYPE_SECURITY_CORE – plik z kodem i danymi, który jest wykonywany w fazie *Security* (*SEC*) procesu rozruchu; faza SEC jest pierwszą fazą procesu rozruchu UEFI;

EFI_FV_FILETYPE_PEI_CORE – plik wykonywalny, który inicjuje fazę *Pre-EFI Initialization* (*PEI*) procesu rozruchu; faza PEI następuje po fazie SEC;

EFI_FV_FILETYPE_PEIM – moduły PEI, będące plikami z kodem i danymi, wykonywanymi w fazie PEI;

EFI_FV_FILETYPE_DXE_CORE – plik wykonywalny, który inicjuje fazę *Driver Execution Environment* (*DXE*) procesu rozruchu; faza DXE następuje po fazie PEI;

EFI_FV_FILETYPE_DRIVER – plik wykonywalny uruchamiany w fazie DXE;

EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER – plik z kodem i danymi, który może być wykonany zarówno w fazie PEI, jak i DXE;

EFI_FV_FILETYPE_ APPLICATION – aplikacja UEFI, która jest plikiem wykonywalnym możliwym do uruchomienia w fazie DXE;

EFI_FV_FILETYPE_FFS_PAD – plik dopełnienia.

W przeciwieństwie do typowych systemów plików używanych w systemach operacyjnych, w których pliki mają nazwy czytelne dla człowieka, pliki FFS są identyfikowane przez identyfikatory GUID.

Poznawanie sekcji plików

Większość plików oprogramowania układowego przechowywanych w FFS składa się albo z jednej części, albo z wielu odrębnych części nazywanych sekcjami (choć niektóre pliki, takie jak *EFI_ FV_FILETYPE_RAW*, nie zawierają żadnych sekcji).

Istnieją dwa typy sekcji: sekcje liści oraz sekcje opakowania. Sekcje liści bezpośrednio zawierają dane, których typ jest określony atrybutem typu sekcji w jej nagłówku. Sekcje opakowań zawierają sekcje plików, które mogą zawierać albo sekcje liści, albo opakowań. Oznacza to, że sekcja opakowania może zawierać zagnieżdżoną sekcję opakowania.

Oto kilka typów sekcji liści. Zawierają one kolejno:

EFI_ SECTION_PE32 – obraz PE;

EFI_SECTION_PIC – kod niezależny od pozycji (PIC);

EFI_SECTION_TE – obraz Terse Executable (TE);

EFI_SECTION_USER_INTERFACE – zawiera łańcuch interfejsu użytkownika; zazwyczaj używany jest do przechowania czytelnej dla człowieka nazwy pliku jako uzupełnienia GUID;

EFI_SECTION_FIRMWARE_VOLUME_IMAGE – opakowany obraz oprogramowania układowego.

A oto para sekcji opakowań zdefiniowanych w specyfikacji FFS:

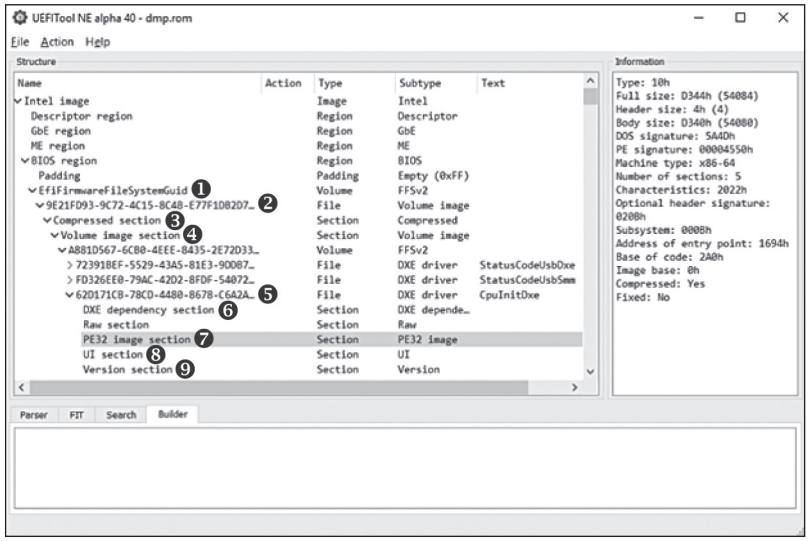
EFI_SECTION_COMPRESSION – zawiera sekcje skompresowanych plików;

EFI_SECTION_GUID_DEFINED – opakowuje inne sekcje zgodnie z algorytmem identyfikowanym przez GUID sekcji; ten typ jest na przykład wykorzystywany dla sekcji podpisanych.

Obiekty te tworzą zawartość oprogramowania układowego UEFI w nowoczesnych platformach. Analityk musi uwzględnić każdy komponent oprogramowania układowego bez względu na to, czy sekcja zawiera kod wykonywalny, taki jak PE32, TE lub PIC, czy też plik danych z nieulotnymi zmiennymi.

Lepsze zrozumienie przedstawionych tu koncepcji umożliwia rysunek 19.17, na którym jest pokazana lokalizacja sterownika `CpuInitDxe` w woluminie oprogramowania układowego. Sterownik ten jest odpowiedzialny za

inicjowanie procesora w fazie DXE. Przejdziemy od niego w górę hierarchii FFS, aby opisać jego lokalizację w obrazie oprogramowania układowego.



Rysunek 19.17. Lokalizacja sterownika CpuInitDxe w regionie BIOS

Obraz wykonywalny sterownika jest zlokalizowany w sekcji obrazu PE32 ❼. Sekcja ta, wraz z innymi, które zawierają nazwę sterownika ❽, wersję ❾ i uwarunkowania ❻, zlokalizowana jest w pliku o GUID wynoszącym `{62D171CB-78CD-4480-8678-C6A2A797A8DE}` ❺. Plik jest częścią opakowanego woluminu oprogramowania układowego ❹ przechowywanego w skompresowanej sekcji ❸. Skompresowana sekcja zlokalizowana jest w pliku `{9E21FD93-9C72-4C15-8C4B-E77F1DB2D792}` ❷ o typie woluminu oprogramowania układowego, który jest przechowywany w woluminie najwyższego poziomu ❶.

Przykład ten miał na celu zademonstrowanie hierarchii obiektów tworzących oprogramowanie układowe UEFI, ale jest to tylko jedno z możliwych podejść jego analizowania.

Teraz, gdy już wiemy, jak zorganizowany jest region BIOS-u, będziemy w stanie poruszać się po jego hierarchii i wyszukiwać różne obiekty przechowywane w oprogramowaniu układowym BIOS-u.

Analizowanie obrazu oprogramowania układowego przy użyciu Chipsec

W tym podrozdziale omówimy analizę oprogramowania układowego przy użyciu platformy Chipsec (*https://github.com/chipsec/*), którą przedstawiliśmy

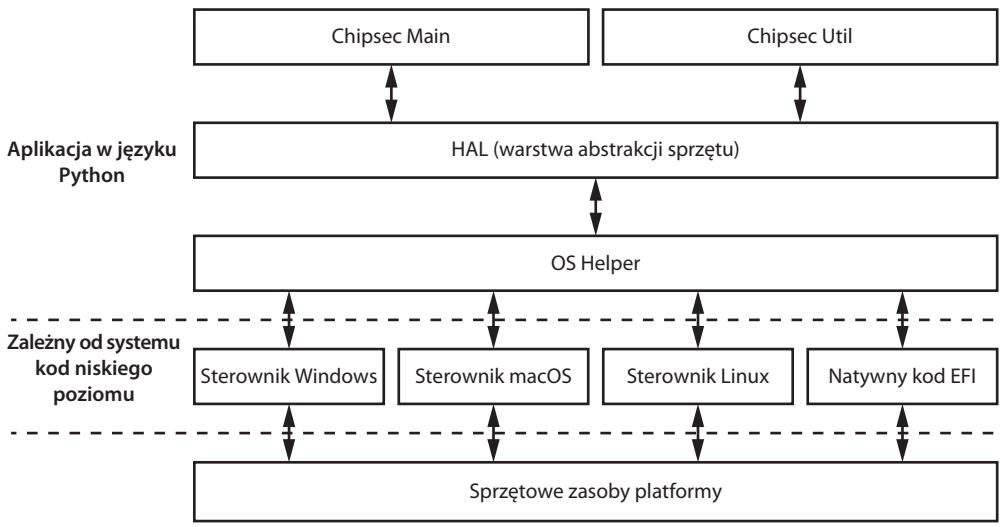
w rozdziale 15. Bardziej szczegółowo zbadamy architekturę tego narzędzia; następnie wykonamy analizę pewnego oprogramowania układowego, przedstawiając kilka przykładów demonstrujących funkcjonalność i przydatność narzędzia Chipsec.

Narzędzie udostępnia wiele interfejsów służących do oceny zasobów sprzętowych platformy, takich jak pamięć fizyczna, rejestry PCI, zmienne NVRAM oraz pamięć flash SPI. Interfejsy te są bardzo użyteczne dla analityka śledczego i przyjrzymy się im poważniej w dalszej części tego podrozdziału.

W celu zainstalowania i skonfigurowania narzędzia należy postępować zgodnie z przewodnikiem instalacji w podręczniku narzędzia Chipsec (*https://github.com/chipsec/chipsec/blob/master/chipsec-manual.pdf*). Podręcznik ten omawia również mnóstwo funkcjonalności, które można wykorzystać, ale w tym podrozdziale skupiamy się tylko na możliwościach analizy śledczej narzędzia Chipsec.

Poznawanie architektury Chipsec

Na rysunku 19.18 jest pokazany schemat architektury narzędzia.



Rysunek 19.18. Architektura narzędzia Chipsec

Na samym spodzie widzimy moduły zapewniające dostęp do zasobów systemu, takich jak mapowane do pamięci zakresy adresów wejścia/wyjścia, rejestry konfiguracji przestrzeni PCI i pamięć fizyczna. Są to moduły zależne od platformy, implementowane jako sterowniki trybu jądra oraz natywny kod EFI. (Obecnie Chipsec udostępnia sterowniki trybu jądra dla systemów

Windows, Linux i macOS). Większość modułów napisanych jest w języku C i mają być wykonywane w trybie jądra lub w powłoce EFI (*shell*).

UWAGA *UEFI Shell jest aplikacją UEFI, która udostępnia interfejs wiersza poleceń dla oprogramowania układowego, pozwalający na uruchamianie aplikacji UEFI i wykonywanie poleceń. UEFI Shell można użyć do odczytywania informacji o platformie, przeglądania i modyfikowania zmiennych menedżera rozruchu, ładowania sterowników UEFI i wielu innych działań.*

Powyżej tych komponentów niskiego poziomu zależnych od systemu operacyjnego znajdziemy warstwę abstrakcji niezależną od systemu, nazywaną OS Helper, złożoną z wielu modułów, które ukrywają przed resztą aplikacji specyficzne dla systemu API komunikacji z komponentami trybu jądra. Moduły zlokalizowane na tym poziomie są zaimplementowane w Pythonie. Od dołu moduły te współdziałają z komponentami trybu jądra; od góry zapewniają interfejs niezależny od systemu operacyjnego dla kolejnego komponentu, czyli warstwy abstrakcji sprzętu (*hardware abstraction layer* – HAL).

HAL zapewnia dalszą abstrakcję niskopoziomowych koncepcji platformy, takich jak rejestry konfiguracji PCI oraz rejestry specyficzne dla modelu (MSR), i zapewnia interfejs dla komponentów Chipsec zlokalizowanych na poziomach bezpośrednio nad nim: *Chipsec Main* oraz *Chipsec Util*. HAL również jest napisana w Pythonie i polega na OS Helper w celu uzyskiwania dostępu do specyficznych dla platformy zasobów sprzętowych.

Dwa pozostałe komponenty, umieszczone na szczycie architektury, zapewniają główną funkcjonalność dostępną dla użytkowników. Pierwszy interfejs, Chipsec Main, dostępny jest przez skrypt Pythona *chipsec_main.py* w głównym folderze narzędzia. Umożliwia wykonywanie testów, które sprawdzają konfigurację zabezpieczeń określonych aspektów platformy, wykonuje PoC w celu przetestowania obecności podatności w oprogramowaniu układowym systemu i wiele innych. Drugi interfejs, Chipsec Util, jest dostępny przez skrypt *chipsec_util.py*. Można go wykorzystywać do uruchamiania indywidualnych poleceń i uzyskiwania dostępu do sprzętowych zasobów platformy, aby odczytywać obraz pamięci flash SPI, zrzucać zmienne NVRAM UEFI itd.

Nas interesuje głównie interfejs Chipsec Util, gdyż zapewnia bogatą funkcjonalność przydatną w pracy z oprogramowaniem układowym UEFI.

Analizowanie oprogramowania układowego przy użyciu Chipsec Util

Polecenia udostępniane przez Chipsec Util można znaleźć, uruchamiając skrypt *chipsec_util.py* zlokalizowany w głównym katalogu repozytorium narzędzia bez specyfikowania żadnych argumentów. Mówiąc ogólnie, polecenia są pogrupowane w moduły według zasobów sprzętowych platformy, których dotyczą. Oto kilka najbardziej użytecznych modułów:

acpi – implementuje polecenia umożliwiające pracę z tabelami Advanced Configuration and Power Interface (ACPI);

cpu – implementuje polecenia związane z procesorem, takie jak odczytywanie rejestrów konfiguracji i uzyskiwanie informacji o procesorze;

spi – implementuje wiele poleceń umożliwiających pracę z pamięcią flash SPI, w tym odczytywanie, zapisywanie i wymazywanie danych; dostępna jest również opcja wyłączenia ochrony zapisu BIOS-u w systemach z odblokowaną ochroną zapisu (zgodnie z omówieniem w rozdziale 16);

uefi – implementuje polecenia pozwalające analizować oprogramowanie układowe UEFI (region BIOS-u pamięci flash SPI) i wydobywać pliki wykonywalne, zmienne NVRAM i temu podobne.

W celu uzyskania dodatkowych informacji można uruchomić `chipsec_util.py` *`nazwa_polecenia`*, gdzie *`nazwa_polecenia`* jest poleceniem, które chcemy poznać. Spowoduje to wyświetlenie opisu i informacji na temat korzystania z danego polecenia. Na przykład, w listingu 19.2 jest pokazane wyjście polecenia `chipsec_util.py spi`.

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.3.3h
[CHIPSEC] API mode: using OS native API (not using CHIPSEC kernel module)
[CHIPSEC] Executing command 'spi' with args []

❶ >>> chipsec_util spi info|dump|read|write|erase|disable-wp
[flash_address] [length] [file]

    Examples:

    >>> chipsec_util spi info
    >>> chipsec_util spi dump rom.bin
    >>> chipsec_util spi read 0x700000 0x100000 bios.bin
    >>> chipsec_util spi write 0x0 flash_descriptor.bin
    >>> chipsec_util spi disable-wp
```

Listing 19.2. Opis i informacje o posługiwaniu się modułem `spi`

Jest to przydatne, gdy chcemy poznać obsługiwane opcje poleceń o takich nazwach, które są samo-objaśniające, jak `info`, `read`, `write`, `erase` lub `disable-wp` ❶. W kolejnych przykładach będziemy używać głównie poleceń `spi` oraz `uefi`, aby pobrać i rozpakować obraz oprogramowania układowego.

Zrzucanie i analizowanie obrazu pamięci flash SPI

Na początek przyjrzyjmy się modułowi `spi`, który pozwala pobrać oprogramowanie układowe. Polecenie to wykorzystuje podejście programowe do zrzucenia zawartości pamięci flash SPI. Aby uzyskać obraz flash SPI, możemy wykonać następujące polecenie:

```
chipsec_util.py spi dump ścieżka_do_pliku
```

gdzie *ścieżka_do_pliku* jest ścieżką do lokalizacji, w której chcemy zapisać obraz SPI. Po udanym wykonaniu polecenia plik ten będzie zawierać obraz pamięci flash.

Teraz, gdy mamy obraz pamięci flash SPI, możemy go analizować i wydobyć użyteczne informacje za pomocą polecenia `decode` (warto wspomnieć, że to samo polecenie `decode` można wykorzystać do analizy obrazu flash SPI uzyskanego metodą sprzętową), jak poniżej:

```
chipsec_util.py decode ścieżka_do_pliku
```

gdzie *ścieżka_do_pliku* wskazuje plik zawierający obraz flash SPI. Chipsec wykona analizę i wydobędzie dane przechowywane w obrazie flash, zapisując je w katalogu. Zadanie to można również wykonać, używając polecenia `uefi` z opcją `decode`, jak poniżej:

```
chipsec_util.py uefi decode ścieżka_do_pliku
```

W każdym przypadku po udanym wykonaniu polecenia uzyskujemy zbiór obiektów wydobytych z obrazu, takich jak pliki wykonywalne, pliki danych ze zmiennymi NVRAM oraz sekcje plików.

Zrzucanie zmiennych NVRAM UEFI

Użyjemy teraz narzędzia Chipsec do wyliczenia i wydobycia zmiennych UEFI z obrazu pamięci flash SPI. W rozdziale 17 pokazaliśmy skrótowo, jak użyć polecenia `chipsec uefi var-list` do wydobycia zmiennych NVRAM. Mechanizm UEFI Secure Boot poleca na zmiennych NVRAM przechowujących dane konfiguracyjne, takich jak wartość zasady Secure Boot, klucz platformy, klucze wymiany kluczy oraz dane `db` i `dbx`. Uruchomienie tego polecenia zwraca listę wszystkich zmiennych NVRAM przechowywanych w obrazie oprogramowania układowego UEFI wraz z ich zawartością i atrybutami.

To tylko kilka poleceń spośród bogatego arsenału narzędzia Chipsec. Wyczerpująca lista wszystkich przypadków użycia Chipsec wymagałaby

oddzielnej książki. Jeśli ktoś jest zainteresowany tym narzędziem, zalecamy zapoznanie się z jego dokumentacją.

To kończy naszą analizę oprogramowania układowego przy użyciu Chipsec. Po wykonaniu tych poleceń uzyskamy wydobytą zawartość obrazu oprogramowania układowego. Kolejnym krokiem analizy byłaby analiza każdego indywidualnego komponentu z wykorzystaniem narzędzi właściwych dla typu uzyskanego obiektu. Na przykład do analizy modułów PEI i DXE możemy wykorzystać deasembler IDA Pro, jednocześnie przeglądając zmienne NVRAM UEFI w edytorze heksadecymalnym.

Ta lista poleceń Chipsec może posłużyć jako dobry punkt wyjścia do dalszego poznawania oprogramowania UEFI. Zdecydowanie zachęcamy do pobawienia się tym narzędziem i lekturę podręcznika, aby poznać jego inne możliwości i funkcje w celu pogłębienia swojej wiedzy na temat analiz śledczych oprogramowania układowego.

Podsumowanie

W tym rozdziale omówiliśmy ważne podejścia do analizy oprogramowania układowego UEFI: uzyskiwanie oprogramowania oraz jego analizę i wydobywanie informacji.

Omówiliśmy dwa różne sposoby uzyskiwania oprogramowania układowego – podejście programowe oraz sprzętowe. Podejście programowe jest wygodne, ale nie zapewnia w pełni wiarygodnego sposobu uzyskania obrazu oprogramowania układowego z systemu docelowego. Z tego względu zalecamy stosowanie podejścia sprzętowego, mimo większej trudności.

Zademonstrowaliśmy również korzystanie z dwóch narzędzi open source, które są nieodzowne w analizowaniu i inżynierii wstecznej obrazów flash SPI: UEFITool oraz Chipsec. UEFITool udostępnia funkcjonalność przeglądania, modyfikowania i wydobywania danych śledczych z obrazu flash SPI, a Chipsec jest przydatny przy wykonywaniu wielu operacji wymaganych w analizach śledczych. Użycie Chipsec pozwala również zrozumieć, jak łatwo napastnik może zmodyfikować obraz oprogramowania układowego złośliwym payloadem. Spodziewamy się, że zainteresowanie badaniami oprogramowania układowego w branży bezpieczeństwa będzie się znacząco powiększać.