

SONY PLAYSTATION 3

Ausnutzen von Sicherheitslücken

Projektdokumentation

zur Anerkennung als Teilleistung
des Moduls „Embedded IT-Security“

vorgelegt von

Timo Christeleit

und

Leon Pascal Olejar

Studienbereich Design Informatik Medien
Hochschule RheinMain

Zeitraum: 19.05.2023 - 28.08.2023

Modulleiter: M.Sc. Fabio Campos

Inhaltsverzeichnis

1 Projektbeschreibung	4
1.1 Einführung und Ausgangslage	4
1.1.1 Vertrauenskette	4
1.2 Projektziele	5
1.3 Ideen und Vorgehensweise	6
1.3.1 ECDSA-Sicherheitslücke	6
1.3.2 Vorgehensweise	7
2 Implementierung	7
2.1 Grundlagen	7
2.1.1 Analyse der Spieldateien	8
2.1.2 Funktionsweise des App-Loaders	12
2.1.3 Entschlüsselung der EBOOT.BIN	12
2.2 Exploit-Umsetzung	13
2.2.1 Entschlüsselung der EBOOT.BIN-Dateien mit dem scetool	14
2.2.2 Übersetzen der entschlüsselten EBOOT BIN Dateien	14
2.2.3 Extrahierung der ECDSA-Signaturen	15
2.2.4 Berechnung des Hashwerts	15
2.2.5 ECDSA-Kurvenparameter	16
2.2.6 Theoretische Berechnung der Zufallszahl	17
2.2.7 Langzeit-Arithmetik	18
2.2.8 Montgomery-Darstellung	19
2.2.9 Praktische Berechnung der Zufallszahl	19
2.2.10 Praktische Berechnung des privaten Schlüssels	20
2.3 Game Modding	22
2.3.1 Modifikationen ohne Private-Key Exploit	22
2.3.2 Modifikationen mit Private-Key Exploit	22
2.3.3 Modifikationen von Maschinencode	23
3 Fazit	24

Abbildungsverzeichnis

1.1 Chain of Trust	5
2.1 Verzeichnis-Struktur: Borderlands und Nascar 09	8
2.2 EBOOT.BIN-Aufbau	9
2.3 Entschlüsselung der EBOOT.BIN	13

1 Projektbeschreibung

1.1 Einführung und Ausgangslage

Die Sony PlayStation 3 ist eine Spielkonsole, die ursprünglich im Jahr 2006 erschienen ist. Sie ist die Nachfolgerin der PlayStation 2 und wurde 2013 von der PlayStation 4 abgelöst. Da Spielekonsolen schon immer ein beliebtes Ziel für Hacker sind, wurden auch bei der PlayStation 3 im Laufe der Jahre verschiedene Sicherheitslücken gefunden und Angriffe durchgeführt, um die Konsole für Homebrew-Software zu öffnen oder um Raubkopien abspielen lassen zu können.

Die Ausgangslage für diese Projektdokumentation basiert auf einem Exploit, der im Dezember 2010 von einer Hackergruppe namens fail0verflow auf dem 27. Chaos Communication Congress vorgestellt wurde. Dieser Exploit nutzt eine Sicherheitslücke in der Implementierung des ECDSA-Algorithmus (Elliptic Curve Digital Signature Algorithm) aus, um die privaten Schlüssel der Konsole zu extrahieren. Mit diesen Schlüsseln ist es möglich, eigene Software zu signieren und auf der Konsole auszuführen. Um die Bedeutung der Sicherheitslücke im Zusammenhang mit der PlayStation 3 besser verstehen zu können, muss die Vertrauenskette (Chain of Trust) und die ECDSA-Implementierung der Konsole genauer betrachtet werden. [\[Sveb\]](#)

1.1.1 Vertrauenskette

Der Boot-Vorgang der Playstation 3 erfolgt in mehreren Stufen. Zuerst wird die Secure-Boot-Phase ausgeführt, die direkt in der Hardware implementiert ist. In dieser Phase wird der Bootloader aus dem Read-Only Memory (ROM) geladen und dessen Integrität überprüft. Der Bootloader wird entschlüsselt und ausgeführt, wobei dieser wiederum den Level-0 Loader lädt, dessen Integrität prüft, ihn entschlüsselt und ausführt. Anschließend liest der Level-0 Loader den Meta-Loader, prüft dessen Integrität, entschlüsselt ihn und führt ihn aus. Der Meta-Loader führt die gleichen Schritte aus und lädt weitere Loader, wie den Level-1 Loader, den Level-2 Loader, den App-Loader, den Isolation Loader und den Revocation-Loader.

Es ist zu erkennen, dass die Sicherheit des Systems auf dieser Vertrauenskette beruht, wobei davon ausgegangen wird, dass der erste Schritt des Boot-Vorgangs nicht manipuliert wird und dass jeder Schritt die Integrität des nächsten Schrittes überprüft, bevor dieser ausgeführt wird. Es wäre nicht möglich z.B. den Meta-Loader zu ersetzen, da die Integritätsprüfung fehlschlägt und der Boot-Vorgang abgebrochen wird.

Um die Vertrauensketten zu durchbrechen, müsste die Wurzel der Vertrauensketten verändert werden, was einen direkten Zugriff auf die Hardware erfordert. Eine zweite Möglichkeit ist die Umgehung der Integritätsprüfung. Diese zweite Option wird von Sony selbst durch einen Fehler in der ECDSA-Implementierung ermöglicht. [Svea]

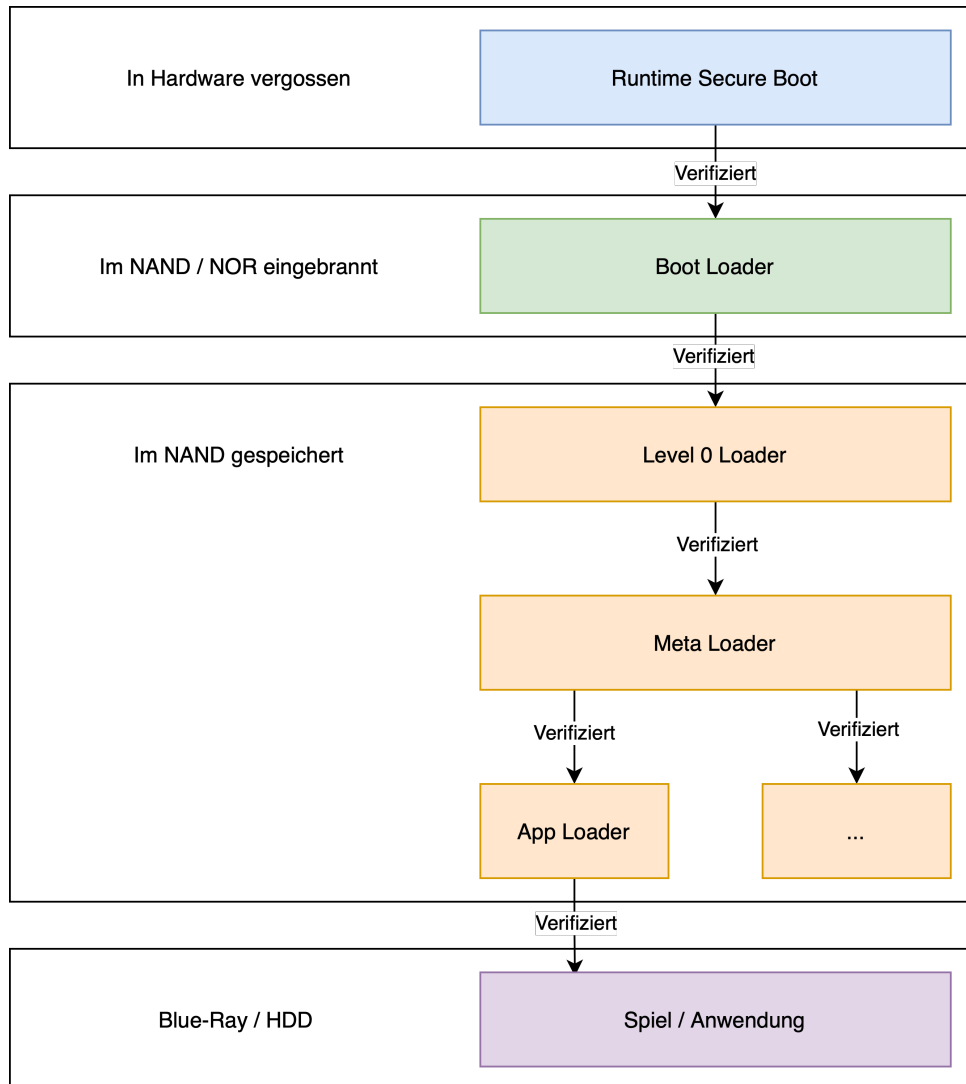


Abbildung 1.1: Chain of Trust

1.2 Projektziele

Das Ziel dieses Projektes ist es, die ECDSA-Implementierung der PlayStation 3 zu analysieren und die Sicherheitslücke zu verstehen, die es ermöglicht, den privaten Schlüssel der Konsole zu extrahieren. Diese Sicherheitslücke soll ausgenutzt werden, um modifizierte Spiele auf der Konsole ausführen zu können.

1.3 Ideen und Vorgehensweise

1.3.1 ECDSA-Sicherheitslücke

Um die Sicherheitslücke zu verstehen, muss die Integritätsprüfung der PlayStation 3 analysiert werden. Die Integritätsprüfung wird mit der sogenannten Elliptic Curve Cryptography (ECC) durchgeführt. Zur ECC gehört auch der ECDSA-Algorithmus. Es handelt sich hierbei um ein asymmetrisches Signaturverfahren d.h. der Urheber bestimmter Daten kann einen privaten Schlüssel verwenden, um eine digitale Signatur für diese Daten zu erstellen und die Daten zusammen mit der Signatur veröffentlichen. Jeder, der im Besitz des öffentlichen Schlüssels ist, kann die Signatur überprüfen, um die Authentizität der Daten festzustellen. Hinzu kommt, dass jeder der den öffentlichen Schlüssel besitzt, zwar Signaturen überprüfen, aber keine Signaturen mit diesem Schlüssel erstellen kann. ECDSA basiert auf der Schwierigkeit des diskreten Logarithmus-Problems und verwendet elliptische Kurven über endlichen Körpern.

Für die Parametergenerierung wird eine elliptische Kurve gewählt:

$$E(p, a, b, q, A)$$

Die Schlüsselgenerierung erfolgt in mehreren Schritten:

1. Wähle einen privaten Schlüssel d mit $0 < d < q$
2. Berechne $B = d \cdot A$
3. Setze den öffentlichen Schlüssel $K_{pb} = (p, a, b, q, A, B)$

Die Signaturerstellung erfolgt ebenfalls in mehreren Schritten:

1. Wähle eine Zufallszahl k_E mit $0 \leq k_E \leq q$
2. Berechne $R = k_E \cdot A$
3. Setze $r = x_R$
4. Berechne $s = k_E^{-1} \cdot (h(x) + d \cdot r) \mod q$

Dem Verifizierer wird die Signatur (r, s) zusammen mit den Daten x und dem öffentlichen Schlüssel K_{pb} übergeben.

Eine Signatur kann wie folgt verifiziert werden:

1. Berechne $w = s^{-1} \mod q$
2. Berechne $u_1 = w \cdot h(x) \mod q$
3. Berechne $u_2 = r \cdot w \mod q$
4. Berechne $P = u_1 \cdot A + u_2 \cdot B$
5. Die Signatur ist gültig, wenn $x_P \equiv r \mod q$

Es ist zu erkennen, dass die Sicherheit von ECDSA stark vom Zufallswert k_E abhängt. Sony hat den Fehler gemacht, dass sie für jede Signatur den gleichen Zufallswert verwendet haben. Dadurch ist es möglich, den privaten Schlüssel d zu berechnen, wenn zwei Signaturen mit dem gleichen k_E bekannt sind. Daher ist die Idee dieser Projektarbeit, zwei Signaturen zu finden, die mit dem gleichen k_E erstellt wurden. Anschließend soll der private Schlüssel d berechnet werden, um modifizierte Spiele auf der PlayStation 3 ausführen zu können. [\[Stö23\]](#)

1.3.2 Vorgehensweise

In der Vertrauenskette ist der App-Loader dazu da, um die Integrität von Spielen und Anwendungen zu überprüfen. Das bedeutet, dass Spiele und Anwendungen nur ausgeführt werden können, wenn sie von Sony signiert wurden. Die Vorgehensweise wäre also die Signaturen von zwei Spielen zu extrahieren und zu überprüfen, ob sie mit dem gleichen k_E erstellt wurden. Falls dies der Fall ist, kann der private Schlüssel d berechnet werden.

2 Implementierung

2.1 Grundlagen

Als Grundlage wird eine PlayStation 3 mit der Firmware Version 4.81 und dem bereits vorinstallierten Linux-Betriebssystem „Red Ribbon“ verwendet. Außerdem werden zwei Spiele benötigt, die vor 2010 veröffentlicht wurden. Für dieses Projekt wurden die beiden Spiele „Borderlands“ und „Nascar 09“ verwendet. Mittels Red Ribbon lässt sich auf die Spieldateien zugreifen. Die Daten auf den Blu-Ray Discs sind verschlüsselt und werden hardwareseitig entschlüsselt. Die zu benötigten Spieldateien heißen EBOOT.BIN und werden vom App-Loader auf ihre Integrität überprüft und anschließend ausgeführt. Die EBOOT.BIN befindet sich im „USRDIR“-Verzeichnis einer Anwendung.

Die generelle Verzeichnis-Struktur einer PlayStation 3 Anwendung ist in Abbildung 2.1 zu sehen. Um die Integritätsprüfung besser zu verstehen, müssen die EBOOT.BIN Dateien und deren Aufbau analysiert werden.

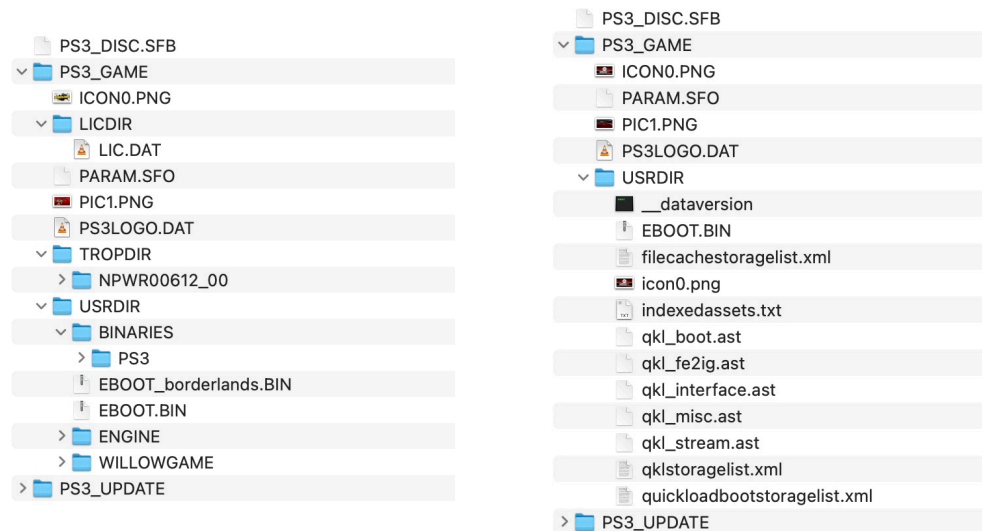


Abbildung 2.1: Verzeichnis-Struktur: Borderlands und Nascar 09

2.1.1 Analyse der Spieldateien

Die EBOOT.BIN wird auch SELF-Datei genannt. SELF steht für Signed Executable and Linkable Format und beschreibt das ausführbare Dateiformat der PlayStation 3. Wie der Name schon verrät sind SELF-Dateien signiert und zusätzlich verschlüsselt. Das Ergebnis einer entschlüsselten SELF-Datei ist eine ELF-Datei. ELF steht für Executable and Linkable Format und ist ein Dateiformat, das für Binärdateien verwendet wird. Sie enthält den ausführbaren Code und die Daten, die für die Anwendungsausführung benötigt werden.

Die EBOOT.BIN besteht aus einem unverschlüsseltem und einem verschlüsseltem Teil. Der unverschlüsselte Teil besteht aus einer Header-Struktur. Die einzelnen Header enthalten spezielle Felder mit entsprechenden Parametern. Die Felder geben Auskunft über die Struktur der Datei oder enthalten sonstige Informationen, die für die Entschlüsselung, Ausführung etc. der Datei benötigt werden. Nur der Certified-File-Header und der Extended-Header sind unverschlüsselt. Alle weiteren Daten sind verschlüsselt.

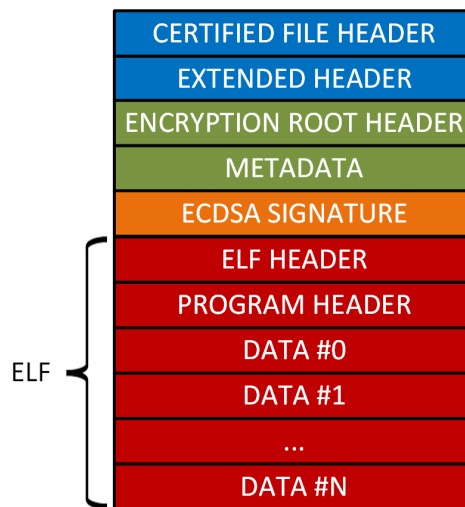


Abbildung 2.2: EBOOT.BIN-Aufbau


Certified File Header:

Der Certified-File-Header ist der erste Header in der EBOOT.BIN und beginnt mit einem Feld namens „magic“. Dieses Feld enthält die Zeichenfolge „SCE\0“ und markiert den Anfang der Datei. Anschließend folgt ein Feld mit der Bezeichnung „version“, das angibt, ob es sich um eine EBOOT.BIN für eine PlayStation 2 oder PlayStation 3 handelt. Es folgen weitere Felder, die unter anderem die Größe der gesamten Datei oder auch das Offset zur ELF-Datei beinhalten.

Extended Header:

Der Extended Header ist geprägt durch Offsets zu allen weiteren Headern. Diese Offsets sind relativ zum Anfang der Datei. Das erste Feld ist wieder ein Versionsfeld, das beschreibt, ob es sich um einen Extended-Header für eine PlayStation 3, PlayStation 4 oder PS Vita handelt. Anschließend folgen Offsets zu diesen Headern:

- Program-Identification-Header
- ELF-Header
- Program-Header
- Section-Header
- Segment-Extended-Header
- Version-Header
- Supplemental-Header

Es folgt ein weiteres Feld, das die Größe des Supplemental-Headers angibt und ein Padding, welches bei den Spielen, die für dieses Projekt verwendet wurden, mit Nullen aufgefüllt ist. Es ist wichtig zu erwähnen, dass der Extended-Header nicht verschlüsselt ist, dafür aber die Header, die in dem Extended-Header referenziert werden. Die referenzierten Header werden im weiteren Kontext als Metadaten bezeichnet, wie es sich auch der Abbildung  entnehmen lässt.

Encryption-Root-Header:

Der Encryption-Root-Header markiert den Anfang des verschlüsselten Teils der EBOOT.BIN. Alle folgenden Daten sind ebenfalls verschlüsselt. Der Encryption-Root-Header enthält einen Schlüssel und einen Initialisierungsvektor (IV), um die nachfolgenden Metadaten zu entschlüsseln. Sowohl der Schlüssel, als auch der IV sind jeweils 128-Bit lang. Es existieren auch sogenannte Fake-SELF Dateien (FSELF). Diese Dateien sind zwar signiert, aber nicht verschlüsselt. Daher entfällt der Encryption-Root-Header und die anschließenden Metadaten und die ELF sind unverschlüsselt. Für dieses Projekt werden allerdings nur die verschlüsselten SELF-Dateien betrachtet.

Metadaten:

Die Metadaten bestehen aus den Headern, die im Extended-Header referenziert wurden. Zusätzlich gehören die folgenden Daten ebenfalls zu den Metadaten:

- Certification-Header
- Certification-Body
- Segment-Certification-Header
- Attributes
- Optional-Header-Table
- Signature

Die Metadaten sind verschlüsselt und werden mit dem Schlüssel und dem Initialisierungsvektor aus dem Encryption-Root-Header entschlüsselt.

ECDSA-Signatur:

Sony hat bei der PlayStation 3 das Signaturverfahren „ECDSA160“ verwendet. Das bedeutet, dass die Signatur und alle dazugehörigen Daten 160 Bit groß sind. Die ECDSA-Signatur befindet sich am Ende von den Metadaten und ist ebenfalls mit dem Schlüssel und dem Initialisierungsvektor aus dem Encryption-Root-Header verschlüsselt. Die Signatur wird benötigt, um die Integrität der EBOOT.BIN zu überprüfen. Deswegen wird eine SHA-1 Hashberechnung über einen Teil der EBOOT.BIN durchgeführt und signiert. Das Ergebnis ist die ECDSA-Signatur. Im Certification-Header befindet sich ein Feld „sig_input_length“, das angibt, über welchen Teil der EBOOT.BIN eine SHA-1 Hashberechnung durchgeführt wird. Im Grunde entspricht dieser Teil der Headerstruktur und den Metadaten aus Abbildung 2.2. Im Certification-Header existiert ein weiteres Feld „sig_offset“, das angibt, an welcher Stelle sich die Signatur in der EBOOT.BIN befindet. Um die Signatur zu extrahieren, müssen die Metadaten zuerst entschlüsselt werden. Anschließend kann die Signatur über das „sig_offset“ im Certification-Header lokalisiert und in die folgende Struktur extrahiert werden:

```

1      typedef struct _signature
2      {
3          uint8_t r[21];
4          uint8_t s[21];
5          uint8_t padding[6];
6      } ECDSA160;
```

ELF:

Wenn eine SELF entschlüsselt wird, dann fallen die vorher beschriebenen Header und Metadaten weg und es bleibt eine ELF-Datei übrig. Sie beinhaltet einen ELF-Header, der die Struktur der ELF-Datei beschreibt. Anschließend folgen die Program-Header, welche die einzelnen Segmente der ELF-Datei beschreiben. Die Segmente enthalten den ausführbaren Code und die Daten, die für die Anwendungsausführung benötigt werden. In den Metadaten befindet sich für jedes Segment der ELF-Datei ein Schlüssel mit einem entsprechendem Initialisierungsvektor, um die einzelnen Segmente zu entschlüsseln.

Näheres zum Aufbau der EBOOT.BIN sowie zu den einzelnen Headern und Metadaten lässt sich unter [\[Wiki\]](#) und [\[Wiki\]](#) finden.

2.1.2 Funktionsweise des App-Loaders

Der App-Loader hat die Aufgabe die EBOOT.BIN zu entschlüsseln und anschließend auszuführen. Dafür wird die Payload der EBOOT.BIN mit dem Advanced Encryption Standard (AES) entschlüsselt. Damit einhergehend werden auch die ECDSA-Signaturdaten entschlüsselt und ein Hash über die Headerstruktur berechnet. Mit dem berechneten Hash wird die ECDSA-Signatur aus den entschlüsselten Metadaten verifiziert. Wenn die ECDSA-Signatur gültig ist, wird die ELF-Datei mit den Schlüsseln und Initialisierungsvektoren aus den Metadaten entschlüsselt. In den Metadaten befinden sich Hashes über die einzelnen Segmente der ELF-Datei. Diese Hashes werden mit den entschlüsselten Segmenten verglichen. Wenn alle Hashes übereinstimmen, wird die ELF-Datei ausgeführt. Für das definierte Projektziel ist es erforderlich die Schritte des App-Loaders zu verstehen und selbst durchführen zu können, um die ECDSA-Signatur extrahieren und verifizieren zu können. Ein wichtiger Faktor hierfür stellt die Entschlüsselung der EBOOT.BIN dar. [\[Wiki\]](#)

2.1.3 Entschlüsselung der EBOOT.BIN

Für die Entschlüsselung der EBOOT.BIN werden zwei AES-Modi verwendet:

- CBC (Cipher Block Chaining Mode)
- CTR (Counter Mode)

Der AES-Algorithmus verwendet eine Blockgröße von 128 Bit und einen Schlüssel mit einer Länge von 128,192 oder 256 Bit, um Daten zu ver- und entschlüsseln. Die PlayStation 3 verwendet für die Entschlüsselung des Encryption-Root-Headers im CBC-Mode einen 256-Bit langen Schlüssel. Dieser Schlüssel wird auch Encryption-Round-Key (ERK) genannt. Zusätzlich wird bei diesem AES-Verfahren ein Reset-Initialisierungsvektor (RIV) von 128-Bit verwendet, um die Sicherheit des Verfahrens zu erhöhen und unterschiedliche Chiffretexte zu ermöglichen, wenn derselbe ERK mehrmals für denselben Klartext verwendet wird. Durch die Entschlüsselung des Encryption-Root-Headers wird der Schlüssel und der IV für die Entschlüsselung der Metadaten extrahiert. Dieser Schlüssel wird auch SELF-Key genannt. Mittels AES-CTR und dem SELF-Key werden die Metadaten entschlüsselt. Durch AES-CTR wird die Blockverschlüsselung (Blockcipher) zu einer Stromverschlüsselung (Streamcipher) umgesetzt. Der besondere Vorteil des CTR-Modus ist der wahlfreie Zugriff auf jeden verschlüsselten Block und die Möglichkeit, sämtliche Ver- und Entschlüsselungsoperationen parallel durchzuführen. Übertragungsfehler wirken sich nur auf den entsprechenden lokalen Block aus. Hinzu kommt, dass nur die Verschlüsselungsvorschrift für die Ver- und Entschlüsselung benötigt wird. Mit der Entschlüsselung der Metadaten geht auch die Entschlüsselung der ECDSA-Signatur einher.

Anschließend werden mit den Schlüsseln und Initialisierungsvektoren aus den Metadaten die einzelnen Segmente der ELF-Datei ebenfalls im AES-CTR entschlüsselt. Es wird eine SHA-1 Hashberechnung über die ELF durchgeführt und mit dem Hash aus den Metadaten verglichen. Wenn die Hashes übereinstimmen, kann die ELF-Datei ausgeführt werden. [\[Ede\]](#)

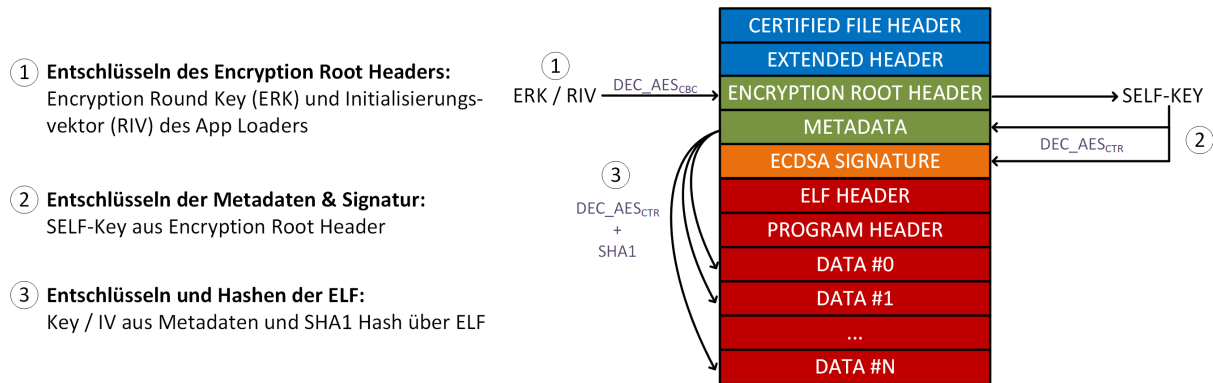


Abbildung 2.3: Entschlüsselung der EBOOT.BIN

2.2 Exploit-Umsetzung

Für die praktische Umsetzung des Exploits müssen die beiden EBOOT.BIN Dateien der Spiele Borderlands und Nascar 09 entschlüsselt und in ein lesbares Format gebracht werden. Die Entschlüsselung der EBOOT.BIN Dateien erfolgt mit dem Open-Source-Tool „scetool“. Dieses ist in der Lage eine EBOOT.BIN zu entschlüsseln und in eine ELF-Datei umzuwandeln. Das scetool implementiert die Entschlüsselung, wie sie im vorherigen Abschnitt beschrieben wurde, in der Programmiersprache C. In dem Gitlab-Repository [\[Lim\]](#), das in den Quellenangaben verlinkt ist, kann der Quellcode für das scetool mit allen Anpassungen und Erweiterungen für dieses Projekt unter „ps3/tools/EBOOT_DECRYPTOR“ gefunden werden.

2.2.1 Entschlüsselung der EBOOT.BIN-Dateien mit dem scetool

Für die Entschlüsselung benötigt das scetool den ERK und den RIV. Diese beiden Parameter werden in der Datei „keys.h“ definiert. Die Datei „keys.h“ ist im Internet zu finden und enthält sämtliche Schlüssel und Initialisierungsvektoren für die verschiedenen Loader der PlayStation 3. Im Gitlab-Repository kann diese Datei gefunden werden.



Ein Problem ist, dass als Ergebnis der EBOOT.BIN Entschlüsselung nur die ELF-Datei bestehen bleibt. Jedoch werden für die Extrahierung der Signaturen, die Offsets und Parameter aus den Metadaten benötigt, um die Signaturen lokalisieren zu können. Das scetool entschlüsselt die gesamte EBOOT.BIN und verwirft die entschlüsselten Daten, die nicht zur ELF gehören. Daher wurde das scetool so erweitert, dass es auch die verworfenen Daten in geeigneten Datenstrukturen speichert. Im Gitlab-Repository sind die Datenstrukturen in der Datei „header.h“ zu finden. Der Dateiaufbau entspricht dem Aufbau der EBOOT.BIN in Abbildung 2.2.

2.2.2 Übersetzen der entschlüsselten EBOOT.BIN Dateien

Die EBOOT.BIN Dateien können mit einem Hex-Editor geöffnet werden. Allerdings müssen im Hex-Editor die Header und Metadaten anhand der einzelnen Bytes interpretiert werden. Da das scetool die Header und Metadaten schon in geeigneten Datenstrukturen speichert, ist es möglich einen Parser zu schreiben, der die Datenstrukturen aus dem scetool ausliest und in eine lesbare Form bringt. Es wurde ein entsprechender Parser geschrieben und dessen Quellcode mit in den Quellcode vom scetool integriert. Die PlayStation 3 verwendet die Big-Endian Byte-Reihenfolge. Der für das Projekt verwendete Rechner verwendet die Little-Endian Byte-Reihenfolge, wie es auch bei den meisten Rechnern der Fall ist. Daher müssen die Datenstrukturen aus dem scetool vom Parser in die Little-Endian Byte-Reihenfolge konvertiert werden, um sie auf dem Rechner lesen zu können. Es werden die bereits vorhandenen Funktionen aus der Bibliothek „endian.h“ verwendet, um die Konvertierung durchzuführen. Der Parser lässt sich mit dem folgenden Befehl ausführen:

```
./scetool -z <EBOOT.BIN> <EBOOT.ELF> » EBOOT.txt
```

Mit diesem Befehl werden folgende Schritte ausgeführt:

1. EBOOT.BIN wird entschlüsselt und in EBOOT.ELF umgewandelt
2. Die Header und Metadaten werden in entsprechenden Datenstrukturen gespeichert
3. Die Datenstrukturen werden in eine lesbare Form gebracht und in die Datei EBOOT.txt geschrieben

Mit dem Parser wurden auch die EBOOT.BIN Dateien von Borderlands und Nascar 09 übersetzt. Die übersetzten Dateien können dem Anhang entnommen werden. Mit der Übersetzung der EBOOT.BIN Dateien wurden auch die ECDSA-Signaturen in den geeigneten Datenstrukturen gespeichert. Somit ist es möglich im nächsten Schritt die Signaturen zu extrahieren.

2.2.3 Extrahierung der ECDSA-Signaturen

Aus den übersetzten Dateien ergeben sich die folgenden ECDSA-Signaturen:

BORDERLANDS

s_1 0x 
 r 0x093BDB9651493F5DC1090CC624CF959C4E11CB53

NASCAR 09

s_2 0x 
 r 0x093BDB9651493F5DC1090CC624CF959C4E11CB53

Es fällt auf, dass das r -Feld bei beiden Signaturen identisch ist. Das bedeutet, dass beide Signaturen mit dem gleichen k_E erstellt wurden. Diese Tatsache lässt sich leicht nachvollziehen, wenn man die Formeln für die Signaturerstellung im Kapitel 1.3.1 betrachtet.

2.2.4 Berechnung des Hashwerts

Im Grunde wird bei der ECDSA-Signatur ein berechneter Hashwert mit dem privaten Schlüssel des Urhebers und einer Zufallszahl k_E signiert. Der Hashwert wird nicht im Klartext in der EBOOT.BIN gespeichert. Ein Signatuerverifizierer erhält ausschließlich die Signatur und die Daten, die signiert wurden und muss den Hashwert über die empfangenen Daten selbst berechnen, um die Signatur verifizieren zu können. Da der Hashwert für die Berechnung der Zufallszahl k_E und dem privaten Schlüssel d benötigt wird, muss dieser erneut berechnet werden.

Der Hashwert wird in der PlayStation 3 mit dem SHA-1 Algorithmus berechnet. Der SHA-1 Algorithmus ist eine kryptographische Hashfunktion, die einen 160-Bit langen Hashwert erzeugt. Heutzutage gilt der SHA-1 Algorithmus als unsicher, da es bereits Angriffe gibt, die eine Kollision erzeugen können. Um den Hashwert zu berechnen, muss zuerst festgestellt werden, über welche Daten der EBOOT.BIN eine SHA-1 Hashberechnung durchgeführt wird. Dafür muss das Feld „sig_input_length“ im Certification-Header betrachtet werden. Dieses Feld gibt an, über welchen Teil der EBOOT.BIN vom Dateianfang aus eine SHA-1 Hashberechnung durchgeführt wird. Ein Problem stellt erneut die Byte-Reihenfolge dar. Es müssen die zu konvertierenden Datenstrukturen nachvollzogen werden. Das scetool bietet die Möglichkeit aus einer ELF-Datei wieder eine SELF-Datei für die PlayStation 3 zu erstellen. Dafür muss das scetool eigene Signaturen erstellen und die damit einhergehende Hashberechnung durchführen. Deswegen wurde im Quellcode vom scetool nachvollzogen, welche Datenstrukturen vor der Hashberechnung in das Big-Endian Format konvertiert werden. Diese Bereichskonvertierung wurde auch auf die beiden EBOOT.BIN Dateien von Borderlands und Nascar 09 angewendet. Dafür wurde die folgende Funktion geschrieben:

```
void getHashtoSIG(sce_buffer_ctxt_t *ctxt)
```

Diese Funktion führt zuerst die Bereichskonvertierung mit den Funktionen aus der Bibliothek „endian.h“ und anschließend eine SHA-1 Hashberechnung über die entsprechenden Daten durch. Für die Berechnung der SHA-1 Hashwerte wurden die Funktionen aus der Bibliothek „sha1.h“ von Polarssl verwendet. Mit der Berechnung der SHA-1 Hashwerte m_1 und m_2 stehen fast alle benötigten Daten für die Berechnung der Zufallszahl k_E und des privaten Schlüssels d zur Verfügung. Es wird zusätzlich noch das Modul q benötigt, das Bestandteil der öffentlichen ECDSA-Kurvenparameter ist.

2.2.5 ECDSA-Kurvenparameter

Im Gitlab-Repository befindet sich im selben Ordner, in welchem die „keys.h“ Datei liegt, eine weitere Datei „ldr_curves“, die die Parameter für die ECDSA-Kurven enthält. Das scetool ermittelt anhand des Dateityps, der Versionsnummer und dem SELF-Typ (Application, Isolated Loader, Revocation Loader) eine Kurvennummer „ctype“. Die Kurvennummer wird verwendet, um die entsprechenden ECDSA-Kurvenparameter aus der Datei „ldr_curves“ zu extrahieren. Die Kurvenparameter werden vom scetool in geeigneten Datenstrukturen gespeichert und können somit ausgelesen werden. Es stehen nun alle Parameter für die Berechnung der Zufallszahl k_E und des privaten Schlüssels d zur Verfügung.

BORDERLANDS

s_1 0x[REDACTED]
 m_1 0x[REDACTED]
 r 0x093BDB9651493F5DC1090CC624CF959C4E11CB53

NASCAR 09

s_2 0x[REDACTED]
 m_2 0x[REDACTED]
 r 0x093BDB9651493F5DC1090CC624CF959C4E11CB53

ECDSA-Kurvenparameter

p 0xB0E7CAFFC8DEEE8A55A376DB4843DDAB2A1F7AE5
 a 0xB0E7CAFFC8DEEE8A55A376DB4843DDAB2A1F7AE2
 b 0xB03BBCC821A5CE9EC2DC1908044C4A4217339EF1
 q 0x00B0E7CAFFC8DEEE8A55A3050D809ADFE38FA01DAB
 A_x 0xAA916D7129BD306A61FB5FD1CDB096D60C14F3B6
 A_y 0x44B66D45D77CC1B48A17871B894CC0CCECD3BB0A

2.2.6 Theoretische Berechnung der Zufallszahl

Für die Berechnung der Zufallszahl k_E lässt sich die folgende Formel in Verbindung mit der Formel für die Signaturerstellung herleiten:

(1) Differenzengleichung bilden

$$s_1 - s_2 = (k_E^{-1} \cdot (m_1 + d \cdot r)) - (k_E^{-1} \cdot (m_2 + d \cdot r)) \mod q$$

(2) Nach k_E umstellen

$$k_E = \frac{m_1 - m_2}{s_1 - s_2} \mod q$$

(3) Berechnung mit der Inversen

$$k_E = (m_1 - m_2) \cdot (s_1 - s_2)^{-1} \mod q$$

Ein Problem stellt die Multiplikation mit dem Inversen von $s_1 - s_2$ und die Reduktion modulo q dar. Modulo-Rechnungen verlangen eine Division. Diese Berechnungen sind aufwendig bei großen Zahlen. Betrachtet man den Zeitaufwand der CPU-Instruktionen für arithmetische Operationen, so ist eine Multiplikation deutlich aufwendiger als eine Addition oder Subtraktion, und eine Division (mit Rest) ist noch zeitintensiver als eine Multiplikation. Ziel ist es, dass diese Berechnungen effizienter durchgeführt werden können.

2.2.7 Langzeit-Arithmetik

Die meisten Prozessoren verfügen über 32-oder 64-Bit-Register und Instruktionen, um den Inhalt von Registern zu addieren, subtrahieren, multiplizieren oder dividieren. Sie verfügen über spezielle Instruktionen, um in einem Register die Bits nach links oder rechts zu verschieben sowie zur bitweisen UND-Verknüpfung. Dadurch lassen sich die folgenden Instruktionen effizient realisieren:

- Multiplikation mit 2^i
- Division durch 2^i (ohne Rest)
- Reduktionen modulo 2^i

Allerdings werden für kryptographische Anwendungen häufig größere Zahlen als 2^{64} benötigt. Daher müssen die Zahlen in mehrere Register aufgeteilt werden. Am Beispiel von diesem Projekt werden 160-Bit lange Zahlen benötigt. Um trotzdem von den Vorteilen der CPU-Instruktionen zu profitieren, werden diese Zahlen in eine entsprechende Radix-Darstellung umgewandelt:

Zahl $x \in \mathbb{N}_0$ zur Basis $b := 2^w$ darstellen:

$$x = \sum_{i=0}^n x_i \cdot b^i$$

- w ist die Prozessorwortlänge, $x_i \in \{0, \dots, b-1\}$ und $n \in \mathbb{N}_0$.

Dadurch lassen sich die arithmetischen Operationen auf die einzelnen Register aufteilen und die CPU-Instruktionen können effizient genutzt werden. Für eine effiziente Multiplikation zweier ganzer Zahlen modulo q wird die Montgomery-Multiplikation verwendet.

[Mai]

2.2.8 Montgomery-Darstellung

Bei der Methode von Montgomery wird statt im Ring \mathbb{Z}_q , in einem isomorphen Ring gerechnet, dabei tritt an die Stelle der normalen modularen Multiplikation die Montgomery-Multiplikation. Durch eine Abbildung werden die Zahlen in die Montgomery-Darstellung überführt. Die Montgomery-Multiplikation lässt sich ohne die bei der normalen modularen Multiplikation erforderliche Division durchführen. Benutzt werden lediglich Addition, Multiplikation und mod- und div-Operationen modulo einer Zweierpotenz. Bei binärer Zahlendarstellung sind diese letzteren Operationen, durch die entsprechenden Prozessorinstruktionen sehr einfach zu realisieren. Das scetool bietet Bibliotheken für die Langzeit-Arithmetik und die Montgomery-Multiplikation. Mit den Funktionen aus den Bibliotheken lassen sich die Zufallszahl k_E und der private Schlüssel d berechnen. Näheres zur Montgomery-Multiplikation ist in [Mai] zu finden.

2.2.9 Praktische Berechnung der Zufallszahl

Das scetool beinhaltet eine Datei „ec.cpp“ in der die Funktionen für die Transformation in die Montgomery-Darstellung und die Montgomery-Multiplikation implementiert sind. Für dieses Projekt wurde in dieser Datei eine Funktion implementiert, die die Zufallszahl k_E und den privaten Schlüssel d berechnet. Folgender Codeausschnitt zeigt die Implementierung zur Berechnung der Zufallszahl k_E :

```

1  void generate_Random_and_Private(u8* s1,u8* s2,u8* m1,u8* m2,u8* R,
   u8* k_E,u8* d_new){
2      [...]
3      //Transformieren der Zahlen in die Montgomery Darstellung
4      bn_to_mon(m1, ec_q, 21);
5      bn_to_mon(m2, ec_q, 21);
6      bn_to_mon(s1, ec_q, 21);
7      bn_to_mon(s2, ec_q, 21);
8      bn_to_mon(R, ec_q, 21);
9
10     //Subtrahieren der Hashwerte
11     bn_sub(m, m1, m2, ec_q, 21);
12
13     //Subtrahieren der Signaturen
14     bn_sub(s, s1, s2, ec_q, 21);
15
16     //Berechnung der Zufallszahl
17     bn_mon_inv(Sinv, s, ec_q, 21);
18     bn_mon_mul(k_E, Sinv, m, ec_q, 21);
19     [...]
20     //Ruecktransformation der Zahlen
21     bn_from_mon(k_E, ec_q, 21);
22 }
```

Im Grunde werden die folgenden Schritte ausgeführt:

1. Übergabe der für die Berechnungen benötigten Parameter an die Funktion:
 - s_1, s_2, m_1, m_2, r, q
 - In k_E wird die Zufallszahl gespeichert und in d_{new} der private Schlüssel
2. Transformation der Zahlen in die Montgomery-Darstellung mit „*bn_to_mon*“
3. Subtraktion der Hashwerte
4. Subtraktion der Signaturen
5. Berechnung der Zufallszahl mit dem Inversen von $s_1 - s_2$ und der Montgomery Multiplikation
6. Rücktransformation der Zufallszahl in \mathbb{Z}_q mit „*bn_from_mon*“

Die Schritte 3 bis 5 implementieren die Formel (3) aus Kapitel 2.2.6

$$k_E = (m_1 - m_2) \cdot (s_1 - s_2)^{-1} \mod q$$

Die Übergabe der Parameter der beiden Spiele an die Funktion ergibt folgende Zufallszahl:

Zufallszahl k_E

0x 

2.2.10 Praktische Berechnung des privaten Schlüssels

Nachdem die Zufallszahl berechnet wurde, kann der private Schlüssel d berechnet werden. Für die Berechnung wird die Formel für die Signaturerzeugung nach d umgestellt:

Formel für die Signaturerzeugung nach d umstellen

$$d = (s \cdot k_E - m) \cdot r^{-1} \mod q$$

Diese Formel gilt es in der selben Funktion wie für die Zufallszahl zu implementieren. Der Codeausschnitt zeigt die Implementierung zur Berechnung des privaten Schlüssels d . Es muss beachtet werden, dass die Berechnung ebenfalls in der Montgomery-Darstellung durchgeführt wird:

```

1  void generate_Random_and_Private(u8* s1,u8* s2,u8* m1,u8* m2,u8*
    R, u8* k_E,u8* d_new){
2      [...]
3      //Berechnung des privaten Schlüssels
4      bn_mon_mul(s1, k_E, s1, ec_q, 21);
5      bn_sub(s1, s1, m1, ec_q, 21);
6      bn_mon_inv(Rinv, R, ec_q, 21);
7      bn_mon_mul(d_new, Rinv, s1, ec_q, 21);
8
9      //Ruecktransformation der Zahlen
10     bn_from_mon(d_new, ec_q, 21);
11 }

```

Es werden die folgenden Schritte ausgeführt:

1. Die zuvor berechnete Zufallszahl k_E wird mit der Signatur s_1 multipliziert
2. Der Hashwert m_1 wird von dem Ergebnis subtrahiert
3. Das Inverse von r wird berechnet
4. Das Inverse von r wird mit der zuvor berechneten Differenz multipliziert
5. Das Ergebnis entspricht dem privaten Schlüssel d in der Montgomery-Darstellung
6. Rücktransformation des Schlüssels in \mathbb{Z}_q mit „*bn_from_mon*“

Das Einfügen der Spieleparameter und der Zufallszahl ergibt folgenden privaten Schlüssel:

Privater Schlüssel d

0x 

Mit dem privaten Schlüssel können nun eigene Signaturen erstellt werden. Es können eigene oder modifizierte Anwendungen erstellt und signiert werden. Der App-Loader verifiziert die Signaturen. Da diese mit dem privaten Schlüssel von Sony erstellt wurden, werden die Signaturen als gültig erkannt und die Anwendungen werden ausgeführt.

2.3 Game Modding

Während Modding in der PC-Spielewelt schon seit Jahren ein fester Bestandteil ist, ist es auf Konsolen nicht so weit verbreitet. Dies liegt an dem vergleichsweise geschlossenen System, welches auf Konsolen herrscht. Weder die Spieleentwickler, noch die Konsolenhersteller wollen, dass die Spiele auf den Konsolen verändert werden können. Dies hat unter anderem den Grund, dass die Spieleentwickler und Konsolenhersteller die Kontrolle über die Spiele behalten und ein Öffnen des Systems zu Sicherheitslücken und Piraterie führen kann.

Trotzdem gibt es mehrere Wege, Spiele auf der Playstation 3 zu modifizieren. Diese können in zwei grundsätzliche Kategorien unterteilt werden.

2.3.1 Modifikationen ohne Private-Key Exploit

Die erste Möglichkeit ist es die Spiele ohne den Private-Key Exploit zu modifizieren. Zwar hat Sony das Sicherheitskonzept der ECDSA-Signatur für die EBOOT.BIN eingeführt, haben aber den Spieleentwicklern die Freiheit gelassen, ab diesem Punkt weitere ausführbare Dateien zu laden und auszuführen. Sind vom Spieleentwickler keine weiteren Sicherheitsmechanismen implementiert worden, können diese Dateien modifiziert werden, ohne vom Sicherheitssystem von Sony erkannt zu werden.

Ein einfaches Beispiel hierfür ist Minecraft. Zwar ist das gesamte Spiel in der EBOOT.BIN enthalten, jedoch werden weitere Dateien geladen die unter anderem die Texturen, Sprachdateien oder sogar DLC's enthalten. So können ohne Probleme Texturen oder Texte ausgetauscht werden.

Durch die Erstellung oder Modifizierung von DLC's können aber auch neue Inhalte hinzugefügt werden, welche auch als Modifikationen genutzt werden können.

2.3.2 Modifikationen mit Private-Key Exploit

Sony hat mit seinem Sicherheitskonzept das Modifizieren der EBOOT.BIN verhindert. Da es sich hierbei um den Einstiegspunkt des Spieles handelt, sind Modifikationen an der Kern-Mechanik des Spieles nicht möglich, ohne diese Sicherheitsmaßnahmen zu umgehen. Durch den Implementierungsfehler in der ECDSA-Signatur ist es jedoch möglich jede beliebige EBOOT.BIN neu zu signieren. Somit können auch Änderungen an diese Datei vorgenommen werden.

Der Einstiegspunkt liegt jedoch immer im Maschinencode vor. Somit ist eine Modifikation des Spieles erschwert. Je nach Spiel werden weitere Dateien geladen, welche einfacher zu dekompile sind, da diese gegebenenfalls in einer höheren Programmiersprache geschrieben wurden und nicht in Maschinencode kompiliert wurden.

2.3.3 Modifikationen von Maschinencode

Die Modifikation von Maschinencode ist sehr aufwendig und erfordert ein tiefes Verständnis des Prozessors. Daher gibt es Tools welche den Maschinencode in Assembler umwandeln können. Dieser ist zwar immer noch sehr komplex, jedoch einfacher zu verstehen. Genannte Tools unterstützen auch oft den Versuch einer Rückübersetzung in eine höhere Sprache wie C oder C++. Dies macht das Analysieren und Verstehen des Codes einfacher. Modifikationen müssen jedoch weiterhin in Assembler vorgenommen werden.

Ghidra ist ein Tool, welches von der NSA entwickelt wurde und 2019 als Open-Source veröffentlicht wurde. Mit diesem Tool ist das zuvor genannte möglich. Durch eine Open-Source Erweiterung ist Ghidra außerdem in der Lage den Einstiegspunkt von PS3 Anwendungen zu finden sowie System Calls zu erkennen und aufzulösen. [\[1\]](#) Dies macht das Analysieren von Spielen deutlich einfacher.

Aber auch Ghidra hat seine Grenzen. So ist die Übersetzung in höheren C Code nur als *Pseudo-Code* zu verstehen und kann nicht direkt kompiliert werden. Eine direkte Modifizierung in C ist somit nicht möglich und die entsprechenden Stellen müssen in Assembler modifiziert werden.

Da die Spiele ohne Debug-Optionen kompiliert wurden, sind auch keine Debug-Symbole enthalten. Somit muss das Spiel mühselig von Hand analysiert und reverse-engineered werden. Im Beispiel von Minecraft handelt es sich hierbei um eine ca. 11 MB große Assembler Datei, welche alle Funktionen des Spieles bereitstellt.

Eine effiziente Modifizierung ist hierbei nur bedingt möglich und erfordert viel Zeit und Wissen über den Maschinencode sowie der Plattform selbst.

3 Fazit

Zusammenfassend lässt sich sagen, dass die Extrahierung der ECDSA-Signaturen aus den EBOOT.BIN Dateien von Borderlands und Nascar 09 und die anschließende Berechnung des privaten Schlüssels erfolgreich war. Trotzdem muss gesagt werden, dass die Sicherheitsmechanismen der PlayStation 3 für damalige Verhältnisse sehr gut waren. Die Vertrauenskette der PlayStation 3 ist sehr komplex und die einzelnen Komponenten sind sehr gut abgesichert. Es ist nicht ohne weiteres möglich, die Sicherheitsmechanismen zu umgehen oder eine bestimmte Komponente zu kompromittieren. Es wird entweder ein Hardware-Exploit benötigt oder das Wissen über die Fehlimplementierung des ECDSA-Algorithmus. Nichtsdestotrotz wird ECDSA auch heute noch verwendet und ist ein wichtiger Bestandteil der Kryptographie. ECDSA wird beispielsweise in vielen Kryptowährungen wie Bitcoin, Ethereum und anderen verwendet, um digitale Signaturen für Transaktionen zu erzeugen und die Integrität der Transaktionsdaten sicherzustellen. Doch auch in Sachen IoT-Sicherheit (Internet of Things) spielt ECDSA eine wichtige Rolle. Die PlayStation 3 ist ein gutes Beispiel dafür, dass die Sicherheit von Systemen nicht nur von der Komplexität der Sicherheitsmechanismen abhängt, sondern auch von deren Implementierung.

Bezüglich der Projektdurchführung lässt sich sagen, dass während der Implementierung des Exploits immer wieder neue Probleme aufgetreten sind. Sei es die Byte-Reihenfolge, die Berechnung des Hashwerts, die Langzeit-Arithmetik oder das Modifizieren einer SELF Datei. Doch mit der Einarbeitung in die Thematik und dem Verständnis für die einzelnen Schritte, ließen sich die Probleme Schritt für Schritt lösen. Die Durchführung des Projekts erwies sich als äußerst lehrreich und interessant. Es wurden viele neue Erkenntnisse gewonnen und ein tieferes Verständnis für die Kryptoverfahren in der PlayStation 3 erlangt.

Literatur

- [cli] clienthax. *PS3 Ghidra Scripts*. URL: <https://github.com/clienthax/Ps3GhidraScripts> (besucht am 26. 08. 2023).
- [Ede] Edepot. *PlayStation 3 Secrets*. URL: <http://www.edepot.com/playstation3.html> (besucht am 17. 08. 2023).
- [Mai] Goethe-Universität Frankfurt am Main. *Effiziente Arithmetik in ZM und Fpn*. URL: <https://www.math.uni-frankfurt.de/~ismi/schnorr/lecturenotes/montgomery.pdf> (besucht am 17. 08. 2023).
- [Stö23] Marc Stöttinger. *Security - Authentizität und Verbindlichkeit*. Hochschule Rhein-Main, 2023.
- [Svea] Calle Svensson. *How a not so random number broke the PS3*. URL: <https://www.youtube.com/watch?v=j6yU9z8mtRE> (besucht am 14. 08. 2023).
- [Sveb] Calle Svensson. *My favorite huge number - script*. URL: <https://docs.google.com/document/d/1riSbs2LRCKK4RwSaMnTzRSDFjgWvQ4galxGvMmwxI2E/edit#heading=h.cxbm91dtax1u> (besucht am 14. 08. 2023).
- [Tim] Leon Pascal Olejar Timo Christeleit. *PS3*. URL: <https://tchri001/ps3> (besucht am 18. 08. 2023).
- [Wika] PS3 Developer Wiki. *Certified File*. URL: <https://www.psdevwiki.com/ps3/Certified File> (besucht am 16. 08. 2023).
- [Wikb] PS3 Developer Wiki. *Loaders*. URL: <https://www.psdevwiki.com/ps3/Loaders> (besucht am 17. 08. 2023).
- [Wikc] PS3 Developer Wiki. *SELF-SPRX*. URL: <https://www.psdevwiki.com/ps3/SELF - SPRX> (besucht am 16. 08. 2023).

Borderlands EBOOT.BIN

Certified Header

Magic:	SCE
Version:	2
Attribute:	1
Category:	1
Extended Header Size:	1040
File Offset:	2432
File Size:	26533824

Extended Header

Extended Header Version:	3
Program Identification Offset:	112
ELF Header Offset:	144
Program Header Offset:	208
Section Header Offset:	26534336
Segment Extended Header Offset:	656
Version Header Offset:	912
Supplemental Header Offset:	960
Supplemental Header Size:	112
Extended Padding:	0

Program Identification

Program Authority ID:	1157425104250994691
Program Vender ID:	16777218
Program Type:	4
Program SCE Version:	281474976710656
Padding:	0

ELF Header

ELF Identification:	7F 45 4C 46 02 02 01 66 00 00 00 00 00 00 00 00
ELF Type:	2
ELF Machine:	21
ELF Version:	1
ELF Entry:	25692280
ELF Program Header Offset:	64
ELF Section Header Offset:	26531904
ELF Flags:	0
ELF Header Size:	64
ELF Program Header Entry Size:	56
ELF Program Header Entry Count:	8
ELF Section Header Entry Size:	64
ELF Section Header Entry Count:	30
ELF Section Header String Index:	29

ELF Program Segment Header

Segment Type:	1
Segment Flags:	4194309
Segment Offset:	0
Segment Virtual Address:	65536
Segment Physical Address:	65536
Segment File Size:	24255328
Segment Memory Size:	24255328
Segment Alignment:	65536

ELF Section Header		
Section Name:		0
Section Type:		0
Section Flags:		0
Section Address:		0
Section Offset:		0
Section Size:		0
Section Link:		0
Section Info:		0
Section Address Alignment:		0
Section Entry Size:		0
Segment Extendend Header		
Segment Extended Offset		2432
Segment Extended Header Size:		24255328
Segment Extended Compression:		1
Segment Extended Unknown:		0
Segment Extended Encryptpion:		0
Version Header		
Version Header Type:		1
Version Header Present:		1
Version Header Size:		48
Version Header Unknown:		0
Encryption Root Header		
Key:		
Key_Pad:		
IV:		
IV_Pad:		
Signatur		
r:		
s:		
Padding:		00 00 00 00 00 00
Hash		
Hash:		

NASCAR 09 EBOOT.BIN

Certified Header

Magic:	SCE
Version:	2
Attribute:	1
Category:	1
Extended Header Size:	1040
File Offset:	2432
File Size:	10583064

Extended Header

Extended Header Version:	3
Program Identification Offset:	112
ELF Header Offset:	144
Program Header Offset:	208
Section Header Offset:	10583512
Segment Extended Header Offset:	656
Version Header Offset:	912
Supplemental Header Offset:	960
Supplemental Header Size:	112
Extended Padding:	0

Program Identification

Program Authority ID:	1157425104250994691
Program Vender ID:	16777218
Program Type:	4
Program SCE Version:	281474976710656
Padding:	0

ELF Header

ELF Identification:	7F 45 4C 46 02 02 01 66 00 00 00 00 00 00 00 00
ELF Type:	2
ELF Machine:	21
ELF Version:	1
ELF Entry:	10229168
ELF Program Header Offset:	64
ELF Section Header Offset:	10581080
ELF Flags:	0
ELF Header Size:	64
ELF Program Header Entry Size:	56
ELF Program Header Entry Count:	8
ELF Section Header Entry Size:	64
ELF Section Header Entry Count:	31
ELF Section Header String Index:	30

ELF Program Segment Header

Segment Type:	1
Segment Flags:	4194309
Segment Offset:	0
Segment Virtual Address:	65536
Segment Physical Address:	65536
Segment File Size:	9972552
Segment Memory Size:	9972552
Segment Alignment:	65536

ELF Section Header		
Section Name:		0
Section Type:		0
Section Flags:		0
Section Address:		0
Section Offset:		0
Section Size:		0
Section Link:		0
Section Info:		0
Section Address Alignment:		0
Section Entry Size:		0
Segment Extendend Header		
Segment Extended Offset		2432
Segment Extended Header Size:		9972552
Segment Extended Compression:		1
Segment Extended Unknown:		0
Segment Extended Encryptpion:		0
Version Header		
Version Header Type:		1
Version Header Present:		1
Version Header Size:		48
Version Header Unknown:		0
Encryption Root Header		
Key:		<div></div>
Key_Pad:		<div></div>
IV:		<div></div>
IV_Pad:		<div></div>
Signatur		
r:		00 09 3B DB 96 51 49 3F 5D C1 09 0C C6 24 CF 95 9C 4E 11 CB 53
s:		<div></div>
Padding:		00 00 00 00 00 00
Hash		
Hash:		<div></div>