Techniken in OpenBSD zur Vermeidung von ROP-Angriffen

Jan Klemkow

genua GmbH
jan_klemkow@genua.de

Zusammenfassung

Dieser Beitrag erläutert neue Sicherheitstechniken im Betriebssystem OpenBSD, welche in den letzten Jahren und Monaten hinzugekommen sind, um sich gegen ROP-Angriffe zu wehren. Zunächst wird noch einmal grundlegend auf die ROP-Thematik eingegangen, um das Problemfeld einzuleiten. Danach werden wechselweise die verschiedene Angriffsmöglichkeiten dargelegt zusammen mit den neuen Sicherheitsfunktionen, die diese Angriffe erschweren oder sogar ganz verhindern. Abschließend wird am Ende des Beitrag erörtert, welchen Einfluss der Einsatz von OpenSource-Software auf die Umsetzung von progressiven Sicherheitstechniken hat.

1 Einleitung

Das Ausnutzen von Buffer-Overflows zum Infiltrieren von fehlerhaften Computer-Systemen ist seit vielen Jahrzehnten ein Problem. Seither hat sich aber die Methodik, wie man diese Schwachstellen ausnutzt, stark verändert. Nachdem sich die Data-Execution-Prevention durchgesetzt hat, verbleibt die Technik des Return-Oriented-Programming [RBSS12] (ROP), um auch heute noch mit klassischen Buffer-Overflows aktuelle Computer-Systeme zu kompromittieren.

Dieser Beitrag zeigt, wie das auf hohe Sicherheit ausgelegte Open-Source-Betriebssystem OpenBSD mit dieser Angriffstechnik umgeht. Es wird dargelegt, welche neuen Speicherschutztechniken in den letzten Jahren implementiert wurden, um solche Angriffe zu vermeiden. Zunächst werden dabei einige Spezialformen von ROP-Angriffen erklärt und danach die Gegenmaßnahmen erläutert.

2 Return-Oriented-Programming

Das Einschleusen von eigenem Programm-Code (auch Shell-Code genannt) durch einen Buffer-Overflow auf dem Stack eines fehlerhaften Programms ist auf modernen Betriebssystemen nicht mehr ohne weiteres möglich. Der Stack-Bereich ist in den Page-Tables durch das Non-Execution-Bit als nicht ausführbar markiert. Dadurch würde das Programm sofort von Prozessor und Betriebssystem beendet werden, sobald versucht würde, innerhalb dieses Speicherbereichs Programm-Code auszuführen.

In der Abbildung 1 ist ein klassischer Buffer-Overflow mit eingeschleustem Shell-Code auf dem Programm-Stack abgebildet. Der Angreifer überschreibt dabei die Rücksprungadresse der fehlerhaften Funktion mit einer eigenen Adresse. Diese Adresse zeigt auf den ebenfalls auf dem

P. Schartner · N. Pohlmann (Hrsg.) · D•A•CH Security 2018 · syssec (2018) 59-67.

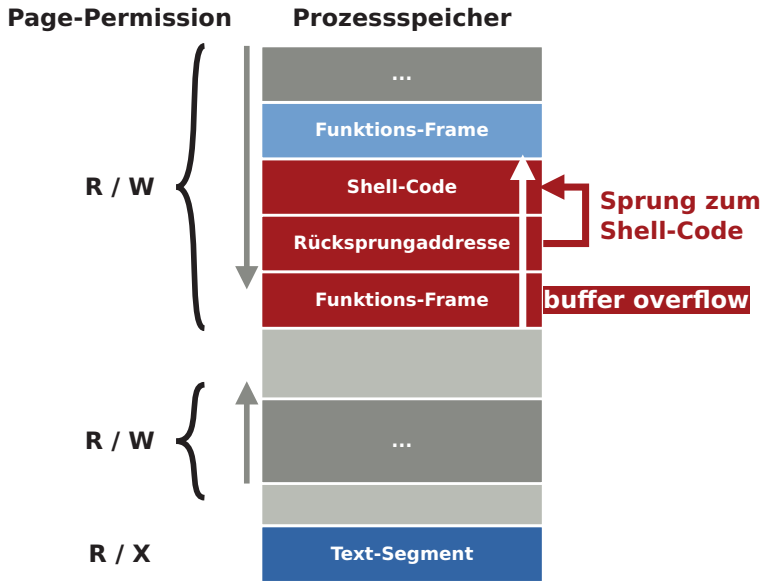

Abb. 1: Schema eines klassischen Buffer-Overflow-Angriffs

Stack eingeschleusten Schadcode (Shell-Code). Sobald der Prozessor versucht, den Programmcode an der Rücksprungadresse zu laden und auszuführen, wird er feststellen, dass dieser in einem Bereich liegt, welcher in den Page-Tables als nicht-ausführbar markiert ist. Daraufhin wird der Prozess unterbrochen und das Betriebssystem über einen Trap informiert. Das Betriebssystem wird dann den Prozess mit einer Fehlermeldung beenden.

Angriffe mittels ROP sind daher in den letzten Jahren zu einer beliebten Methode geworden, um weiterhin die Kontrolle über ein fehlerhaftes Programm zu erlangen.

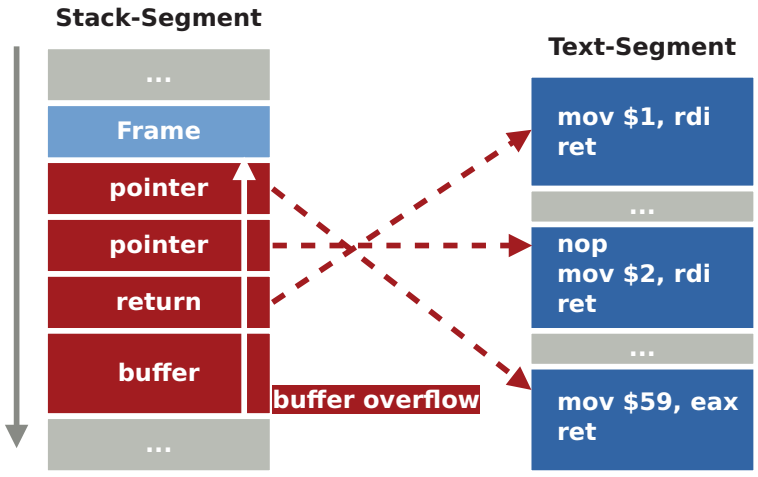

Abb. 2: Schema eines ROP-Angriffs

Da das Einschleusen von eigenem Programm-Code direkt nicht mehr möglich ist, stützen sich ROP-Angriffe darauf, Code-Schnipsel zu verwenden, welche das zu exploitende Programm bereits im eignen Text-Segment enthält (Abbildung 2). Die für einen ROP-Angriff verwendeten Code-Schnipsel werden auch “Gadgets” genannt. Ziel des Angriffs ist es also, den Program-

mablauf zu beeinflussen. Dieses gelingt dem Angreifer, indem er den Stack des Programms mit eigenen Rücksprungadressen überschreibt, welche dann nacheinander abgearbeitet werden. Die Gesamtheit der angesprungenen Code-Stücke bilden dann den Schadcode des Angreifers, ohne, dass dieser explizit eingeschleust werden muss.

3 ASLR

Um es dem Angreifer zu erschweren, die korrekten Adressen von Gadgets anzuspringen, wurde die Address-Space-Layout-Randomization (ASLR) implementiert. Dabei werden verschiedenen Bestandteile eines Prozesses mit zufälligen Offsets in den virtuellen Speicherraum gemapped. Diese Offsets erschweren es dem Angreifer, die gewünschten Gadgets zu treffen.

Die Speichersegmente von Text, Stack und Heap sind nun mit zufälligen Offsets im Speicher angeordnet. Allerdings unterliegen die Offsets der Anforderung, dass die verschiedenen Segmente page-alignt im Speicher liegen müssen. Dadurch kann der Angreifer den Bereich, indem die gewünschten Gadgets liegen stark eingrenzen. Dieses erlaubt es ihm gezielt, sich mit mehreren Angriffen iterativ an die richtigen Speicheradressen heranzutasten, bis sein Angriff funktioniert. Außerdem ist es einem Angreifer möglich, auf die Adressen aller Elemente eines ASLR-Blocks zu schließen, sobald ihm die Adresse eines Objekts innerhalb des Blocks bekannt wird. Dieses kann durch Debug-Ausgaben oder anderen Informations-Leaks geschehen.

4 Zufälliges Re-Linking

Experimente mit dynamischem Linken in zufälliger Reihenfolge hat es auch schon zuvor in der OpenBSD-Entwickler-Gemeinde gegeben [Sha10]. Nur wurden diese erst in den letzten Veröffentlichungen des Projektes produktiv umgesetzt. Dieser Abschnitt erläutert die neuen Sicherheitsmechanismen in diesem Bereich.

4.1 Standard-C-Bibliothek

Return-to-LibC ist ein spezieller ROP-Angriff, welcher sich zueigen macht, dass in fast jedem Programm die Standard-C-Bibliothek eingebunden ist. Diese bietet eine Fülle an ROP-Gadgets, welche für einen Angriff genutzt werden können. Somit kann der Angreifer einen einmal vorhandenen Schadcode-Pfad innerhalb der C-Bibliothek bei Angriffen unterschiedlicher Programme wiederverwenden. Da die C-Bibliothek selbst sehr stabil ist, muss ein Angreifer nur die Position der Bibliothek im Speicher herausfinden. Mit dieser Position kann er dann die genauen Adressen der Gadgets selbst berechnen.

Die Abbildung 3 zeigt, wie dieses im Build- und Release-Prozess realisiert wurde [dR16a]. Zunächst werden die C-Quelldateien zu Shared-Objects kompiliert. Anstatt diese nun direkt in zu dynamischen Bibliothek zu linken, werden diese Dateien in das Archiv “libc.so.a” zusammen gefügt. Dieses Archiv wird in der Binär-Distribution des System ausgeliefert. Bei jedem Start des Systems, wird dieses Archiv entpackt und die enthaltenden Objekt-Dateien in einer zufälligen Reihenfolge zusammen gelinkt. Somit hat jede laufende OpenBSD-Instanz eine eigene Standard-C-Bibliothek, in der die Speicheradressen der ROP-Gadgets individuell verschoben sind.

Ein Angreifer hat dadurch einen enorm höheren Aufwand, die Positionen der benötigen Gadgets zu finden. Es reicht nun nicht mehr, den Offset der Standard-C-Bibliothek zu finden, sondern die Adressen jedes verwendeten Gadgets.

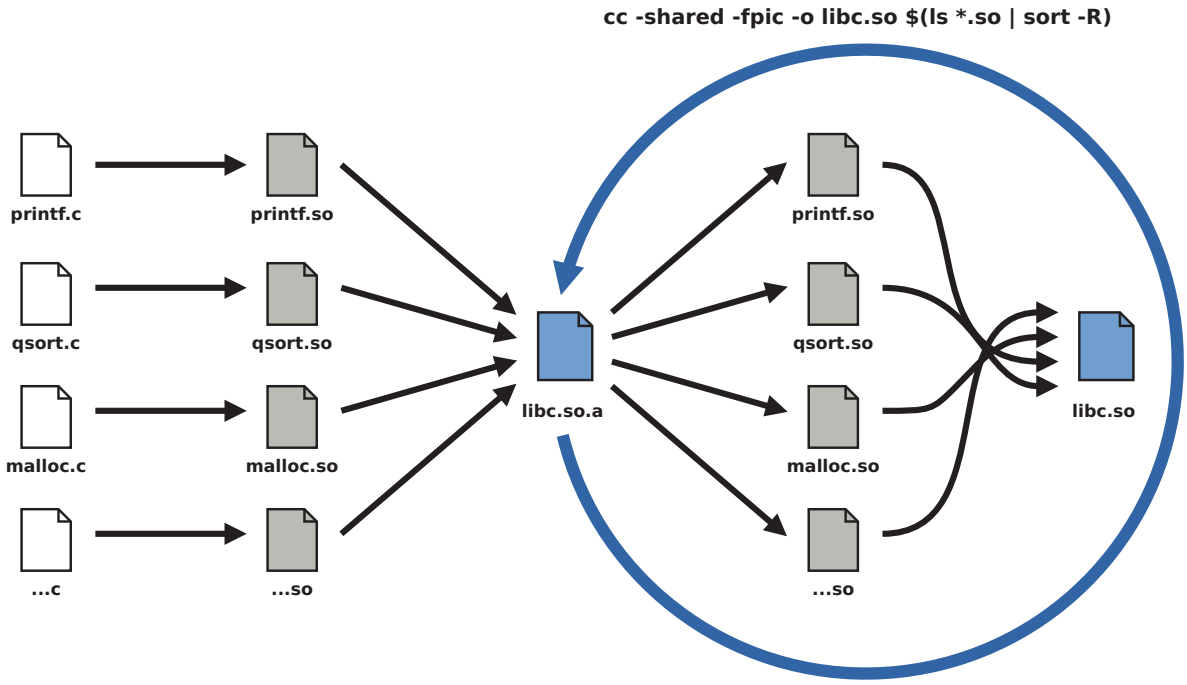

Abb. 3: Relinking der LibC

4.2 Kernel

Diese Technik wird nicht nur bei der Standard-C-Bibliothek angewendet. Um ROP-Angriffe auf den Kernel ebenfalls zu erschweren, wird auch dieser bei jedem Systemstart in zufälliger Reihenfolge neu zusammengelinkt. Ein Problem besteht darin, dass dieses nicht beim Systemstart erfolgen kann, da bereits ein Kernel laufen muss, um das Linker-Programm im Userland auszuführen. Es gibt insgesamt vier Situationen, in denen der Kernel neu gelinkt wird.

Installation: Ebenso wie bei der C-Bibliothek wird auch der Kernel nur noch als Archiv von Objekt-Dateien ausgeliefert. Diese werden nach einer Installation neu zusammengelinkt. Damit wird sichergestellt, dass auch ein frisch installiertes System einen Kernel hat, bei dem ein entfernter Angreifer die Zusammensetzung des Kernels nicht kennt.

Booten: Zudem wird bei jedem Boot der Kernel neu zusammengelinkt. Somit ist nicht nur jeder installierte Kernel einzigartig, sondern jede laufende Instanz.

Update: Und ebenso verhält es sich bei einem Update. Auch hier müssen die aktualisierten und binär-distributierten Objekt-Dateien neu zusammengelinkt werden, damit der aktuelle Kernel gleich beim nächsten Start von dem ROP-Schutz profitieren kann.

Entwicklung: Selbst wenn ein Entwickler oder Anwender sich einen eigenen Kernel aus den Quellendateien baut, sorgt das Buildsystem dafür, dass dieser zufällig zusammengelinkt wird. Somit sind auch experimentelle Test-Systeme mit einem angepassten Kernel geschützt.

4.3 Secure-Boot

Ein zur Zeit noch ungelöstes Problem stellt hier das Zusammenspiel dieses Mechanismus mit Secure-Boot da. Da ein Kernel, welcher immer wieder neu zusammengesetzt wird, auch eine immer neue Prüfsumme hat, ist es für den Bootloader nicht möglich dessen Prüfsumme vor dem Laden gegen eine Signatur zu prüfen. Ein möglicher Lösungsansatz ist, dass der Boot-Loader selbst einen Linker enthält und die einzelnen Objekt-Dateien signiert sind. Damit wäre der Boot-Loader in der Lage, die Signaturen der Objekt-Dateien zu prüfen und anschließend den Kernel zufällig zusammenzulinken. Allerdings würde dieses Vorgehen, die Komplexität des Boot-Loaders enorm erhöhen. Eine abschließende Meinung zu diesem Thema ist zu diesem Zeitpunkt innerhalb der OpenBSD-Entwicklergemeinde noch nicht gefunden. Zum aktuellen Zeitpunkt werden allerdings die Vorteile des zufälligen Linkes als wichtiger angesehen, als den potentiellen Sicherheitsgewinn durch Secure-Boot.

5 Trap-Sled

Damit Programm-Code im Speicher aligned liegt und schneller in den Prozessor-Speicher geladen werden kann, füllt der Compiler die Programm-Segmente mit NOP-Instruktionen (No-Operation) auf. Dieser Padding-Bereich zwischen den regulären Instruktionen wird im normalen Programmablauf nie verwendet. Sollte der Programmablauf dennoch in diese Zwischenbereiche springen, wird der Prozessor die NOP-Instruktionen zusammen mit den dahinter liegenden Instruktionen ausführen. Wobei die NOPs keinerlei Einfluss auf die Programmausführung haben.

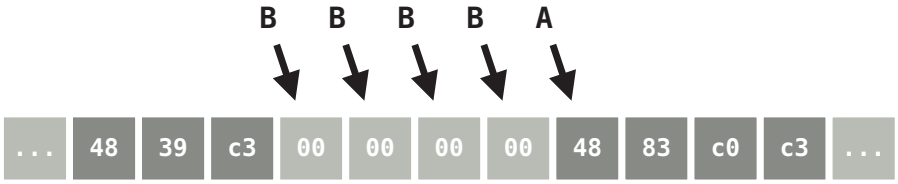

Abb. 4: NOP-Padding

Dadurch bietet sich für einen Angreifer der Vorteil, dass er beim Treffen von Gadgets nicht zwingend den Anfang der Gadgets selbst treffen muss, wenn vor dem Gadget mehre NOPs liegen. Es reicht, wenn der Pointer vom Angreifer auf eine NOP-Instruktion vor dem eigentlichen Gadget zeigt. Im Beispiel aus Abbildung 4, führt jede Einsprungstelle “B” automatisch zur Ausführung der Instruktionen ab der Einsprungstelle “A”. Die NOP-Instruktionen (Hexadezimal: `00`) werden vom Prozessor dann ignoriert, bis die eigentlichen Instruktionen ausgeführt werden. Vereinfacht ausgedrückt, rutscht das Angriff auf den NOPs in den gewollten Gadget-Code.

Um einem Angreifer diese potentiellen Einsprungstellen zu nehmen, wurde der Compiler im OpenBSD-Buildsystem angepasst [dR17]. Bei einem Padding im Text-Segment werden nun keine NOP-Operationen eingefügt, sondern Trap-Instruktionen. Diese Traps sorgen beim Ausführen dafür, dass der Prozessor die Programmausführung unterbricht und das Betriebssystem das Programm beendet. Somit schrumpft der Raum an Möglichkeiten, die einem Angreifer bleiben, um Einsprungstellen zu finden und es steigt der Aufwand, um einen erfolgreichen Angriff durchzuführen.

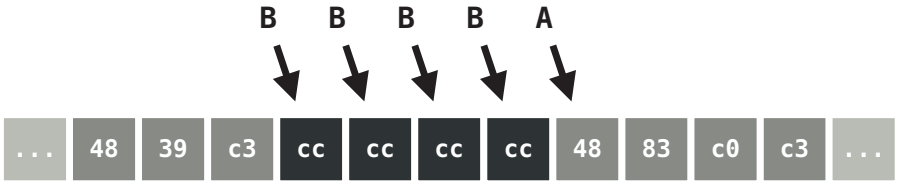

Abb. 5: Trap-Padding

6 RBX-Register vermeiden

Auf Prozessorarchitekturen, die kein striktes Instruktions-Alignment erfordern, kann eine Instruktion an jeder beliebigen Byte-Position beginnen. Für den Aufruf von Gadgets stehen dem Angreifer daher nicht nur direkt die Instruktionen bereit, welche als solche ins Text-Segment kompiliert wurden, sondern auch Daten und Adressen, welche als Instruktionen erscheinen.

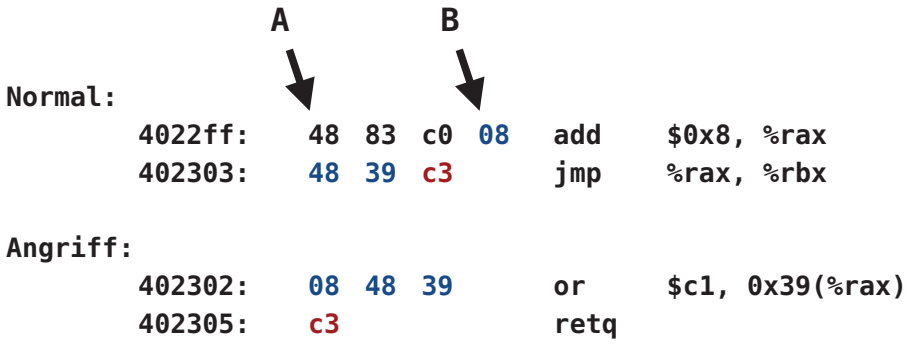

Abb. 6: Unterschiedliche Interpretation von Byte-Code

In der Abbildung 6, ist ein kleines Code-Beispiel gegeben. Im Normalfall werden die Instruktionen vom Prozessor an der Einsprungstelle “A” gelesen und ausgeführt. Bei einem erfolgreichem ROP-Angriff, kann der Angreifer die Adresse der Einsprungstelle auch um drei Byte verschieben. Bei der Ausführung der Byte-Folge ab der Einsprungstelle “B” durch den Prozessor ergeben sich dann, ganz andere Instruktionen.

So hat der Byte-Wert des Registers “RBX” den gleichen Wert, wie der Byte-Wert der Instruktion “ret”. Da die Instruktion “ret” für ROP-Gadgets essenziell ist, lohnt es sich, die Verwendung dieses Byte-Wertes zu vermeiden.

Der Compiler des OpenBSD-Buildsystems wurde dahingegen optimiert, dass er die Benutzung der RBX-Registers vermeidet [Mor17]. Dieses wird umgesetzt, indem der Compiler alle anderen Register gleicher Größe vorzieht und das RBX-Register erst als letztes verwendet. Somit tauchen weniger potentiell nutzbare Gadgets im Text-Segment von Programmen auf, ohne dabei die Performance des erstellten Programms negativ zu beeinflussen.

7 Signal-ROP

Signal-ROP [BB14] ist eine Spezialform von ROP, bei der das Verhalten des Kernels beim Systemcall `sigreturn(2)` ausgenutzt wird. Der `sigreturn(2)` wird eigentlich dafür verwendet, den Zustand eines Prozesses nach der Abarbeitung eines Signals wiederherzustellen.

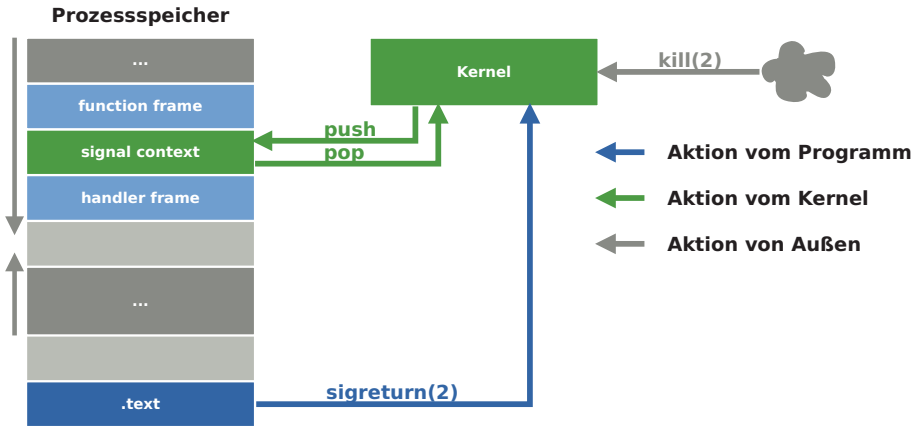

Abb. 7: Signalbehandlung

Abbildung 7 zeigt die reguläre Verwendung des Systemcalls `sigreturn(2)` bei der Signalbehandlung eines Prozesses. Der Kernel sichert vor dem Aufruf der Signalbehandlungsfunktion den Kontext des Programms auf dessen Stack. Nachdem die Signalbehandlungsfunktion abgearbeitet wurde, ruft das Programm den Systemcall `sigreturn(2)` auf. Dieses veranlasst den Kernel, den zuvor gesicherten Programmkontext vom Stack zu lesen und das Programm in den Zustand vor der Signalbehandlung zu versetzen.

Diesen Mechanismus kann ein Angreifer ausnutzen, indem er beispielsweise durch einen Buffer-Overflow einen eigenen Programmkontext auf den Stack legt und mittels ROP den Systemcall `sigreturn(2)` aufruft.

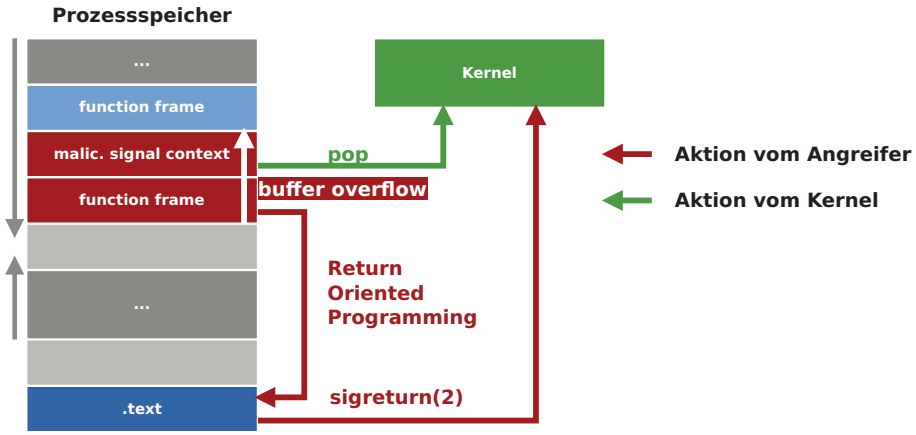

Abb. 8: Schema eines Signal-ROP-Angriffs

Da der Kernel bisher nicht in der Lage war, einen eigenen Signalkontext von einem fremden zu unterscheiden, wurde diese Datenstruktur durch ein Secret erweitert [dR16b]. Erkennt der Ker-

nel beim Aufruf von `sigreturn(2)`, dass dieses Secret verändert wurde, wird der laufende Prozess sofort gestoppt, um Angriffe auf Basis dieser Technik zu unterbinden.

8 Verbesserte Stack-Canaries

Eine übliche Methode zum Erkennen von einfachen Buffer-Overflows auf dem Stack, sind Stack-Canaries. Dabei wird ein Secret, das sogenannte Canary, beim Aufruf einer Funktion in dessen Funktions-Frame direkt vor die Return-Adresse geschrieben. Nach dem Ablauf der Funktion wird überprüft, ob sich dieses Secret auf dem Stack verändert hat.

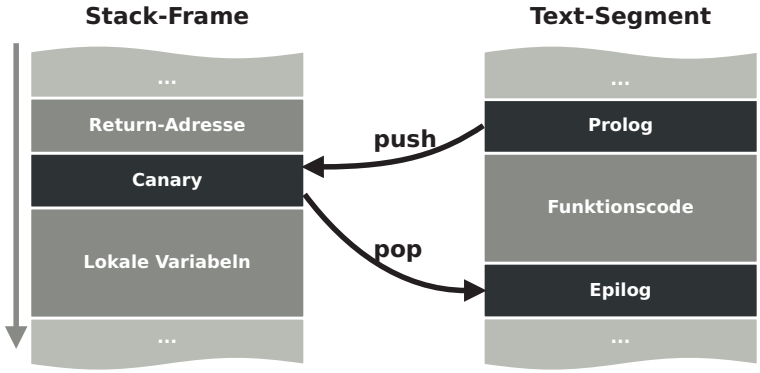

Abb. 9: Verwendung eines Stack-Canary

Wie in Abbildung 9 dargestellt, beginnt jede Funktion mit einem Prolog, welcher im allgemeinen den Stack-Frame vorbereitet. Dabei wird ein globales und nur zur Laufzeit bekanntes Secret als Canary zwischen der Return-Adresse und den lokalen Variablen geschrieben. In dem danach ausgeführten Funktionscode, kann es durch Programmierfehler zum Overflow eines Buffers in den lokalen Variablen kommen, welcher die Return-Adresse überschreibt. Dabei würde zwangsweise auch das Canary überschrieben. Der Funktions-Epilog, welcher den Stack-Frame aufräumt, überprüft dabei, ob das Canary noch den selben Wert hat, wie das globale Gegenstück. Stellt der Epilog einen Unterschied fest, wird die weitere Ausführung des Programms sofort beendet.

Bei einem ROP-Angriff, muss der Angreifer diese Schutzfunktion überwinden, da es sonst nicht zur Ausführung der Gadgets kommt. Das Secret muss dem Angreifer verborgen bleiben. Sollte es ihm gelingen, etwa durch einen Format-String-Angriff, das Canary einer Funktion auszulesen, kann er mit diesem auch andere fehlerhafte Funktionen des selben Prozesses erfolgreich ausnutzen.

Um einen solchen Angriff in Zukunft zu erschweren, wurde die Verwendung von Stack-Canaries für nächste Version (6.4) von OpenBSD angepasst. Es wird nun für jeden Prozess nicht nur ein Secret für Canaries geben, sondern bis zu 4096 verschiedene Secrets. Somit hat praktisch jede Funktion ein eigenes Canary, dessen Leak an einen Angreifer nicht dazu führt, dass dieser auch Fehler in anderen Funktionen ausnutzten kann.

9 Resümee

Durch die Umsetzung der hier geschilderten Verteidigungstechniken, setzt das OpenBSD-Projekt erneut Maßstäbe im IT-Sicherheitsbereich und unterstreicht damit seine Vorreiterrolle als sicherheitsorientiertes Betriebssystem.

Dem OpenBSD-Projekt stehen hier gewisse Freiheiten offen, die andere Betriebssysteme nicht haben. Der Umstand, dass das gesamte Software-Ökosystem aus OpenSource-Software besteht, erlaubt es immer wieder größere Brüche der ABI (Application-Binary-Interface) in Kauf zu nehmen. Viele der entwickelten Sicherheitsfunktionen, bringen fehlerhafte Programme zum Absturz, auch wenn diese nicht direkt angegriffen wurden. Da aber der Source-Code des Ökosystems, inklusive aller 3rd-Party-Programme, offen liegt, ist man im OpenBSD-Projekt in der Lage, diese fehlerhaften Programme zu korrigieren.

Andere Betriebssysteme, wie etwa macOS und Windows, welche in ihrem Ökosystem mit 3rd-Party-Software konfrontiert sind, die fast ausschließlich in Binärform distributiert werden, haben diese Freiheiten nicht. Diese Betriebssysteme müssen in einem weit höherem Maße Abwärtskompatibilitäten unterstützen, um nicht größere Teile ihres Software-Ökosystems zu verlieren, da diese nicht von den Betriebssystem-Entwicklern selbst angepasst und korrigiert werden können.

Durch diese Freiheit ist das OpenBSD-Projekt in der Lage, schnell und unabhängig auf neue Sicherheitsprobleme zu reagieren und kann auch eher akademische oder experimentelle Problemlösungen am eigenen Ökosystem erproben und produktiv umsetzten.

Literatur

[BB14] E. Bosman, H. Bos: Framing signals-a return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 243–258. IEEE, 2014.

[dR16a] T. de Raadt: Anti-ROP mechanism in libc, April 2016.

[dR16b] T. de Raadt: Srop mitigation, May 2016.

[dR17] T. de Raadt: Trapsled, June 2017.

[Mor17] T. Mortimer: Avoid ebx/rbx, November 2017.

[RBSS12] R. Roemer, E. Buchanan, H. Shacham, S. Savage: Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[Sha10] M. Shalayef: Security enchancements in ld(1). Presented at the Easterhegg, Munich, April 2010.