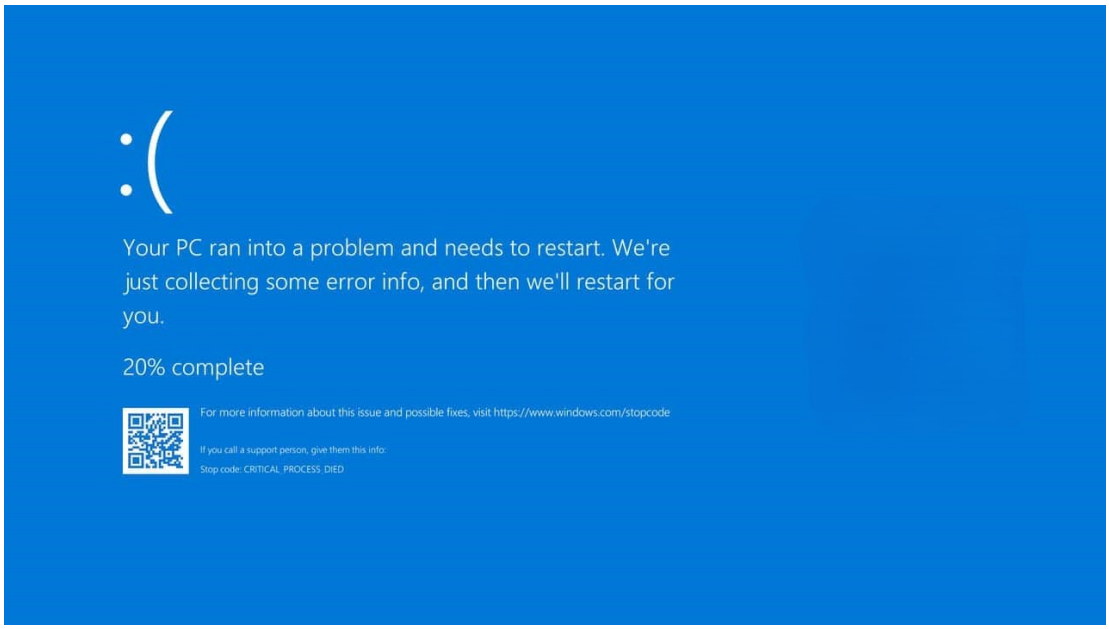
Hector Seminar

DHBW
Duale Hochschule
Baden-Württemberg

Binary Exploitation

Abschlussarbeit der Kooperationsphase 2023/2024

Emilio Craff Castillo, Hendrik Dasselaar



Mit Unterstützung von Markus Götzl von der Dualen Hochschule Baden-Württemberg sowie unserem Kursleiter Thomas Knecht

Inhaltsverzeichnis

Abstract

In computers, a buffer overflow occurs when a program writes more data to a buffer (a designated memory area) than it is intended to hold. This can result in adjacent memory areas being overwritten, potentially leading to crashes, data losses or even, if the overwritten part is exactly planned, the execution of potentially malicious code which can lead to very dramatic scenarios. Therefore, vulnerable or insufficiently coded programs which can cause Buffer Overflows are critical weaknesses, even in modern computers. This documentation examines the Stack Overflow as representative for Buffer Overflows as well as countermeasures like Stack Canaries, ASLR, NX-Bit and others. The heyday of Buffer Overflows has been over since about 2005 because of the complexity of the 64-bit-enhanced x86-architecture which since then has been by far the most used architecture in computers to this day. This is because the 64-bit-enhancement has an extremely high amount of security features built in as standard to prevent Buffer Overflows. Although Buffer Overflows do not happen as often as 30 years ago, they still occur and can be exploited, especially as they commonly appear in the nowadays frequently used programming languages C and C++. As the world is more digitalized than ever, security weaknesses causing Buffer Overflows can theoretically infect millions of computers.

1 Einleitung

Buffer Overflows (dt. Pufferüberläufe) sind ernstzunehmende Schwachstellen in der Computersicherheit, die meistens auf einfache Weise ausgenutzt werden können und zu schwerwiegenden Konsequenzen wie einem Programm- oder Systemcrash, Datenverlust oder im schlimmsten Fall zu einem dauerhaften Verlust der Kontrolle über das System führen können. Sie sind heutzutage nicht mehr so weit verbreitet, wie sie es zu ihren Hochzeiten zwischen 1985 und 2005 waren. Dennoch sind sie immer noch ein wichtiger Angriffspunkt für Hacker und stellen häufig potenzielle Schwachstellen in der Cybersecurity, dem digitalen „Beschützen“ von Servern, Computern und Computernetzwerken sowie Programmen dar. [21]

Die große Gefahr von Buffer Overflows ergibt sich durch ihr Potential, mit dem richtigen Programmierwissen aus nur kleinen Sicherheitslücken heraus ganze Systeme einnehmen zu können. Gefährlich macht sie außerdem die große Masse an Informationen, die online aufgrund ihrer früheren Bekanntheit zu finden sind. Es passiert immer wieder, dass Buffer Overflows mit anderen Sicherheitslücken kombiniert werden, um einen komplexeren Angriff zu auszuführen, als das bei der Ausnutzung von nur einer Sicherheitslücke der Fall wäre. Präventive Maßnahmen und Abwehrtechniken sind aus diesen Gründen für moderne Systeme unerlässlich und müssen sich stetig weiterentwickeln, um auch vor neuen, unbekannten Angriffen durch Ausnutzung von Sicherheitslücken geschützt zu sein und neu erkannte Lücken zu schließen.

Da der Begriff „Buffer“ für alle Arten von temporären Speichern steht und all diese potentiell überlaufen können, gibt es viele unterschiedliche Arten von Buffer Overflows, wie den Integer Overflow, den Format String Overflow, den Heap Overflow oder den Stack-based Buffer Overflow. Der Stack-based Buffer Overflow, in der Kurzform auch Stack Overflow, wird dabei am häufigsten für Buffer Overflow Exploits (dt. sinngemäß mutwillige Ausnutzung der Schwachstelle zum Vorteil des Angreifers oder zum Nachteil des Benutzers) benutzt und ist aufgrund der kritischen Rolle des Stacks innerhalb der Computerarchitektur gleichzeitig auch die gefährlichste Art von Overflows.

Ziel des Projektes ist, den Stack Overflow anhand eigener schematischer Zeichnungen zu erklären und bekannte Sicherheitsmaßnahmen aufzuführen sowie diese mit eigenem C-Programm beispielhaft zu evaluieren. Dafür wird der grundsätzliche Ablauf des Exploits beschrieben. Mehrere gängige und häufig implementierte Sicherheitskonzepte wie ASLR, DEP, NX-Bit und Stackguard sind hier erläutert und zusammengefasst, aber auch Methoden, diese zu umgehen, wie ROP, ret2libc oder GOT Overwrite.

2 Begriffserklärungen

Diese Sektion dient zur Erklärung von Begriffen, die in dieser wissenschaftlichen Arbeit häufig verwendet werden und dessen Kenntnis für eine gute Verständnis beim Lesen unabdingbar ist.

2.1 Festplattenspeicher

Der Festplattenspeicher ist der Bereich eines Computers, auf dem das Betriebssystem sowie alle anderen benutzerbezogenen Daten gespeichert sind. Der von der Festplatte beherbergte Speicher kann entweder eine sich im Inneren drehende HDD (engl. Hard Disk Drive) oder eine SSD (engl. Solid State Drive) sein. Diese Art des Speichers wird auch Non-Volatile Memory (dt. nicht flüchtiger Speicher) genannt, da die auf den Datenträger geschriebenen Daten auch beim Entfernen der an der Festplatte anliegenden Betriebsspannung erhalten bleiben. In heutigen Systemen wird ein Festplattenspeicher von 512 GB für den Großteil der Endnutzer empfohlen. Verwandte Begriffe: Hard Disk Storage (engl. Festplattenspeicher), Festplattenplatz, Speicherplatz, ungenau: Speicher

2.2 Arbeitsspeicher

Der Arbeitsspeicher ist der Bereich des Computers, auf dem während des Hochfahrens des Computers alle Daten von der Festplatte, von der das gerade aktive Betriebssystem stammt, geschrieben werden, die während der Benutzung des Computers häufig benötigt werden bzw. deren schnelles Abrufen wichtig ist. Der Arbsitsspeicher ist, ähnlich wie auch SSDs, immer Flash-basiert, da so die knapp 20- bis 50-fache Geschwindigkeit der Festplatte erreicht wird (heute knapp 140 GB/s für DDR6-17600 vs. 5 GB/s für PCIe 4.0SSDs [19, 20]) und wird auch Volatile Memory (dt. flüchtiger Speicher) genannt. Bei einem etwaigen Stromausfall, einem kompletten Ausschalten bzw. Herunterfahren des Computers oder anderen Situationen, in denen die Spannung unter die Betriebsspannung fällt, bleiben die darauf geschriebenen Daten nicht erhalten und müssen erneut auf den Arbeitsspeicher geschrieben werden. In heutigen Systemen wird ein Arbeitsspeicher von 16 GB für den Großteil der Endnutzer empfohlen. Verwandte Begriffe: RAM (engl. Abk. für Random Access Memory), Programmspeicher, ungenau: Speicher

2.3 „Speicher“

Der Begriff „Speicher“ wird meistens im Umgangssprachlichen verwendet und ist wissenschaftlich nicht korrekt. Dies ist deshalb problematisch, da sowohl der Festplattenspeicher und der Arbeitsspeicher als „Speicher“ bezeichnet werden können. Daher kann der Begriff „Speicher“ nur korrekt verwendet werden, wenn davor bereits entweder der Begriff Festplattenspeicher oder Arbeitsspeicher gefallen ist und sich mit der Benutzung von „Speicher“ eindeutig darauf bezogen wird. Um Verwechslungen vorzubeugen, wird sich deshalb hier so oft wie möglich ausdrücklich auf Arbeits- oder Festplattenspeicher bezogen.

2.4 Register

Register lassen sich vereinfacht als winzig kleine Arbeitsspeicher (64 Bytes in x86-64-basierten Betriebssystemen) verstehen. [6] Sie sind sehr nahe an der CPU (engl. Central Processing Unit, dt. sinngemäß Hauptrecheneinheit) lokalisiert und um ein Vielfaches schneller als der Arbeitsspeicher. Sie können alles Mögliche enthalten, hier einige wenige Beispiele:

Adressen im RAM

Ergebnisse von mathematischen Berechnungen

Einzelne Zahlen und Zeichen

Aktuelle Tastatureingaben etc.

In x86-64-basierten Betriebssystemen gibt es 18 verschiedene Register. Zehn davon sind für spezifische Zwecke, z.B.:

RIP (Instruction Pointer): Enthält die Adresse der als nächstes von der CPU auszuführenden Adresse.

RAX (Accumulator Register): Wird für aritmethische Rechenaufgaben verwendet (Grundrechenarten).

RBP (Base Pionter): Zeigt die Aktuelle Adresse des obersten Stack Frames (siehe 2.5) an.

RFLAGS (Flaggenregister): Enthält Statusflags, die zum Beispiel Ergebnisse von arithmetrischen Operationen sein können.

Zusätzlich dazu hat x86-64 als Erweiterung von x86 acht weitere sogenannte Allzweckregister, R8-R15. Diese können für die meisten Zwecke verwendet werden, die die anderen Register auch erfüllen, außer die der Spezialregister RIP und RFLAGS. Durch die Allzweckregister werden komplexere Operationen bzw. mehr davon gleichzeitig erlaubt, weswegen ein Kern einer x86-64-CPU mit selber Taktfrequenz schneller arbeitet als der einer reinen x86-CPU. [4]

2.5 Function Calls und Rücksprungadresse

Funktionen sind ein Programmierkonzept, welches mehrere einfache Basismethoden zu einem komplexen Gebilde kombiniert und das Programm durch ihre Wiederverwendbarkeit strukturiert. Diese Funktionen befinden sich im sogenannten Codesegment des Arbeitsspeicherbereichs, welcher für ausführbare Dateien reserviert ist. Ein Function Call ist die Art und Weise, wie eine Systemarchitektur das Aufrufen von Funktionen behandelt. Die CPU ruft Funktionen immer nacheinander auf. Die Adresse der nachfolgenden Funktion wird im Register RIP, dem Instruction Pointer, gespeichert. Wenn eine Funktion eine andere aufruft, also einen Function Call durchführt, wird der Inhalt des Instruction Pointers auf dem neuen Stack-Frame, kurz auch Frame (dt. sinngemäß Befehls- bzw. Stapelblock), als Return Adress (dt. Rückgabeadresse) gespeichert. Die Rücksprungadresse (in RIP gespeichert) eines Stack-Frames ist diejenige, die nach erfolgreichem Durchlaufen eines Befehlsblockes wieder zu der Adresse im Arbeitsspeicher zeigt, bei der die nächste Funktion liegt, sodass die CPU nach Beendigen der gerade ablaufenden Funktion das Frame entfernen und zur aufrufenden Funktion zurückkehren kann.

2.6 Brute Force

Brute Force ist eine Methode, einen gesuchten Code, Byte etc. durch reines Ausprobieren herauszufinden. Das heißt, man testet jeden einzelnen möglichen Code auf Richtigkeit. Im Alltag wäre das zum Beispiel das Öffnen eines Zahlenschlosses durch Ausprobieren aller möglichen Zahlenkombinationen. Brute Force wurde in der 32-Bit-Architektur häufig erfolgreich benutzt, seit 64-Bit-Architekturen ist Brute Force jedoch aufgrund der schier unendlichen Anzahl an Kombinationen, die sich aus 64 Bit ergeben, (eine Zahl mit 20 Ziffern,) nicht mehr möglich. [4]

3 Buffer

Buffer (dt. Puffer) sind Bereiche im Arbeitsspeicher, die verwendet werden, um einzelne Daten zu speichern und entsprechen meistens einem sehr kleinen Bereich im gesamten Arbeitsspeicher. Ein Buffer kann alles Mögliche beinhalten und beliebig groß sein. Hier einige wenige Beispiele (in Klammern häufige Größen):

Globale und lokale Variablen für das System oder für andere Programme (1-100 Bytes)

Eingabefelder, die User Input (dt. Benutzereingabe) speichern (1-100 Bytes)

Komprimierte Dateien, wie auch Videos und Fotos (hunderte KB bis wenige GB)

Netzwerkpakete, die der Computer nach ihrem Eintreffen weiterverarbeitet (1-100 KB)

Keyboard-Buffer, also Tastatureingaben, die noch nicht vom System verarbeitet wurden (wenige Bytes)

3.1 Buffer Overflow

Es kommt zu einem Buffer Overflow, wenn die Masse an Daten, die auf einem Buffer gespeichert werden sollen, die Kapazität des Buffers übersteigt, was dazu führt, dass das Programm andere, benachbarte Arbeitsspeicheradressen überschreibt. Dies wird insofern kritisch, wenn die benachbarten Speicheradressen bereits verwendet werden und durch einen Overflow korrumpiert werden, das heißt, dass sie durch einen nicht authorisierten bzw. nicht so vorgesehenen Befehl so mit Daten, sei es versehentlich oder absichtlich, überschrieben werden, dass sie unbrauchbar werden und bei Ausführung zu unerwarteten Ergebnissen führen können (siehe 3.2 und 4.2). [26]

Wenn beispielsweise ein Buffer darauf ausgelegt ist, Benutzereingaben mit einer maximalen Größe von 32 Bytes zu beinhalten, aber mit einem Input von 40 Bytes beschrieben wird, führt das ohne entsprechende Sicherheitsmaßnahmen dazu, dass die letzten 8 Bytes hinter die eigentlich dafür vorgesehene Speicheradresse geschrieben werden und andere Daten überschrieben werden, die teilweise system- und stabilitätsrelevant sein können.

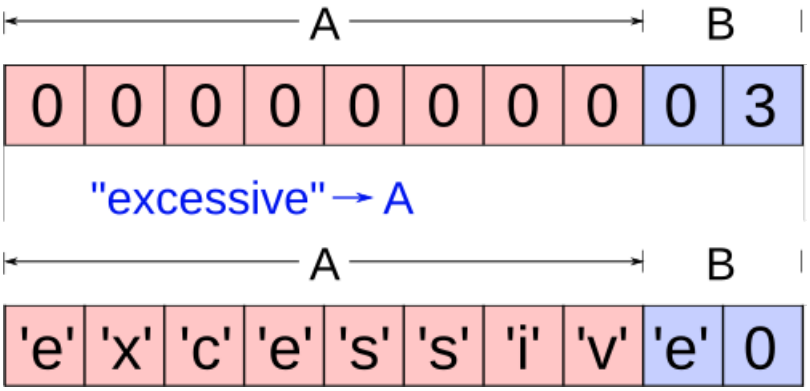


Abb. 1: Die Eingabe „excessive“ ist zu groß für den reservierten Bereich und der Buffer läuft über

Mit Buffer Overflows können Angreifer zum Beispiel Programmierfehler oder fehlende Sicherheitsmaßnahmen auszunutzen und somit Zugriff zu einem System erlangen. Wenn sie die Position bestimmter Daten im Arbeitsspeicher kennen, können sie durch genau passende Eingaben den Speicherbereich hinter dem Buffer gezielt überschreiben. Sie überschreiben zum Beispiel ausführbare Methoden oder Argumente mit eigenen Methoden oder Argumenten, um diese dann auszuführen (siehe 4.3). Durch mehr und bessere Sicherheitsmaßnahmen (siehe 5.) und sich stark verkomplizierende Prozessorarchitekturen wie die 64-Bit-Erweiterung der x86-Architektur werden Buffer Overflow Exploits jedoch immer aufwendiger. [3, 16, 21, 24]

3.2 Vereinfachtes Beispiel eines Buffer Overflows

Dieses Beispiel wird mit selbsterstellten Skizzen vereinfacht dargestellt und soll einen einfachen Einstieg ermöglichen. Es stellt keinen echten RAM dar, da der RAM eines jeden Computers, selbst von uralten PCs, erheblich komplizierter aufgebaut ist und mit Skizzen kaum verständlich erläutert werden kann.

Der Arbeitsspeicher eines fiktiven, vereinfachten Computers besteht aus 32 Megabyte, kurz MB. Die ersten 8 MB des Speichers werden vom System und Systemprogrammen verwendet, weitere 4 MB für beispielsweise ein Textprogramm, welches den letzten seiner 4 MB für das gerade bearbeitete Textdokument reserviert. Danach folgen 8 MB für ein Sicherheitsprogramm, wie hier ein Antivirusprogramm.

Das Antivirusprogramm hat eine sehr hohe Priorität und viele Berechtigungen im System, da es für die Abwehr von Schadsoftware etc. eben diese Berechtigungen braucht, um sich gegen diese durchzusetzen. Also kann die Beeinträchtigung bzw. Korrumption des Antivirusprogrammes schwerwiegende Folgen für das System haben.

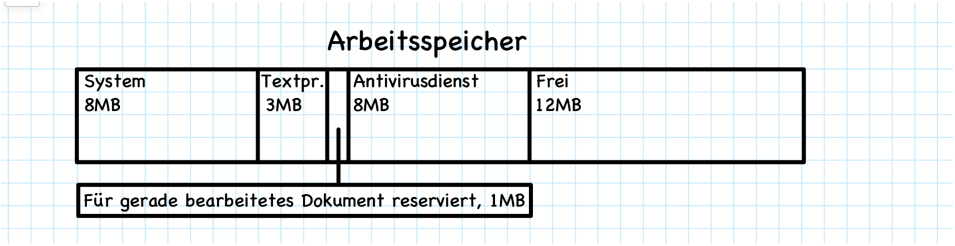


Abb. 2: Aufteilung des fiktiven Arbeitsspeichers

Die Textverarbeitung des Textprogrammes funktioniert vereinfacht so, dass alle Tastatureingaben in Binärcode umgerechnet und in den für das Dokument freigehaltenen Platz einfügt werden.

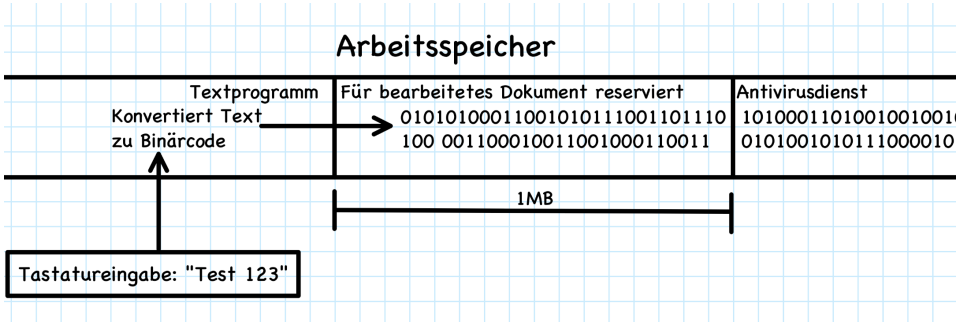


Abb. 3: Vereinfachte, schematische Darstellung der Textverarbeitung

Wenn dieses gerade bearbeitete Dokument aber länger als ca. 100 Seiten und damit größer als 1 MB Arbeitsspeicher wird, ist es zu groß für den reservierten Speicherbereich.

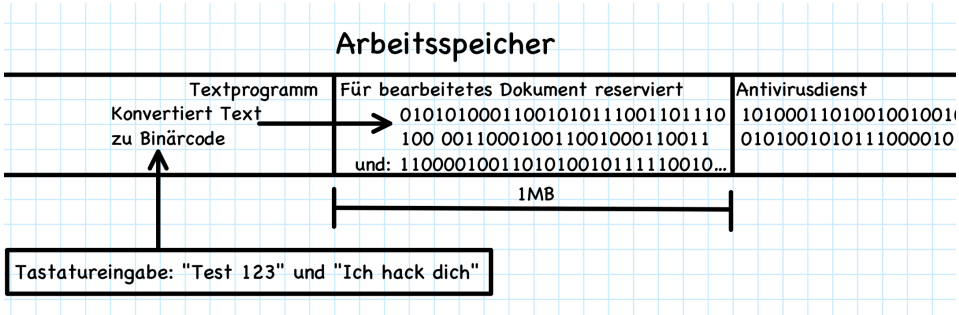


Abb. 4: Textprogramm fügt die zu Code verarbeitete Eingabe in den reservierten Platz ein

Deshalb überschreibt alles, was über den reservierten einen MB hinausgeht, den vom Antivirusprogramm benutzten Arbeitsspeicher, sofern der Computer keine Sicherheitsmaßnahmen besitzt, die ein weiteres sicheres und fehlerfreies Betreiben ermöglichen würden (siehe 5.).

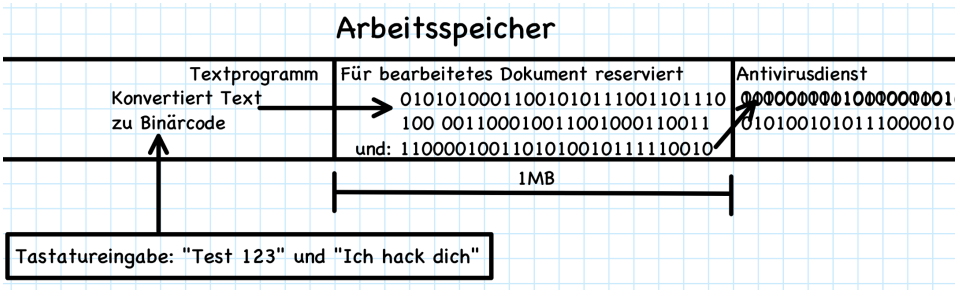


Abb. 5: Das Textprogramm überschreibt den Anfang des Antivirusprogramms

Ein Teil des vom Antivirusdienst verwendeten RAM befindet sich jetzt also in einem anderen Zustand, als er sich eigentlich befinden sollte. Der gesamte Antivirusdienst ist also im jetzigen Zustand korrumpiert (wieder sofern der Computer keine Sicherheitsmaßnahmen besitzt, die diesen überschriebenen Teil erkennen, das betroffene Programm absichtlich abstürzen lassen, wieder von der Festplatte laden und fortsetzen würden).

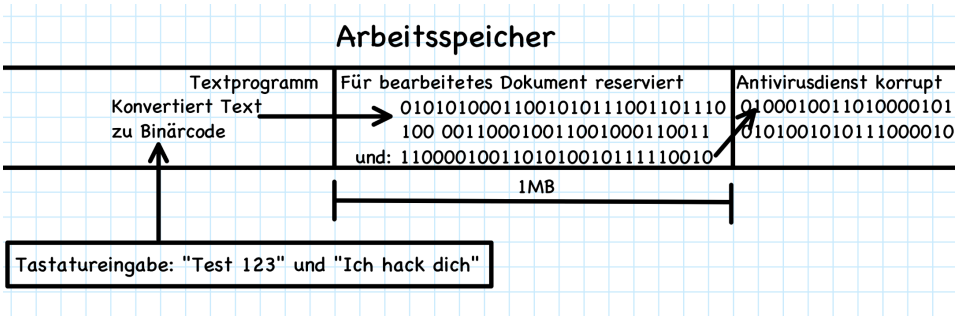


Abb. 6: Korrumpiertes Antivirusprogramm

Sobald dieser Teil des überschriebenen Arbeitsspeichers vom Prozessor oder einer sonstigen Systemkomponente abgerufen wird, führt das (in den vom Benutzer unabsichtlich verursachten Fällen) meistens zu einem Programmabsturz oder (seltener) zu einem Systemcrash. In sehr seltenen Fällen beinhaltet der zu Code kompilierte Text des Textprogrammes durch Zufall einen für Systemprozesse oder andere Low-Level-Prozesse reservierten Befehl und kann Schlimmeres verursachen, zum Beispiel Datenverlust, oder (noch seltener) Korruption von Systemdaten und (extrem selten) damit einhergehende Unbrauchbarkeit des ganzen Systems (sofern noch dieselbe Systeminstallation verwendet wird) und Verlust aller benut-

zerbezogenen Daten, die auf dieser Partition der Festplatte gespeichert waren. Also muss man im allerschlimmsten Fall eine neue Systeminstallation auf die Festplatte durchführen.

3.3 Vergleich des Beispiels mit echten Computern

Dieser Overflow lässt sich nicht als spezieller Stack (Bereich des RAM für kleine Funktionen oder Adressen), Heap (Bereich des RAM für große Funktionen, die ihre Größe während der Laufzeit verändern sowie gerade verwendeten größeren Dateien) oder anderer Overflow kategorisieren, sondern nur als vereinfachtes Beispiel zum leichteren Verständnis. Dies hat mehrere Gründe. Erstens ist der RAM komplizierter aufgebaut als in diesem Beispiel. In echten, modernen Computerm ist er weit von der hier gezeigten Linearität entfernt. Zweitens besitzt bereits die x86-Architektur ohne 64-Bit-Erweiterung zahlreiche Schutzmechanismen, die solch einen simplen Overflow verhindern würden.

4 Stack

Der Stack (engl. für Stapel) ist eine lineare Datenstruktur im RAM. Diese funktioniert im übertragenen Sinne wie ein Bücherstapel: Wenn der Computer neue Datenpakete, Variablen und Befehle, also Stack Frames (siehe 2.5), zum Stack hinzufügt, landen sie oben auf dem Stack, genauso, wie man Bücher auf einen Bücherstapel legt. Ein Frame besteht dabei aus der Return-Adresse, Argumenten und lokalen Variablen. Wenn man nun auf ein bestimmtes Buch, beim Computer also ein bestimmtes Frame, zugreifen will, muss man zuerst all das weglegen, was darüber liegt; das heißt, der Computer muss alle darüber liegenden Frames in umgekehrter Abfolge abarbeiten. Der Stack funktioniert also nach dem LIFO-Prinzip (Last in, First out), der als letztes hinzugefügte Befehl muss also zuerst abgearbeitet werden.

Durch die Natur der Prozessorarchitekturen wird beim Entfernen eines Stack-Frames der RAM-Bereich des Frames nicht gelöscht, (also mit Nullen überschrieben,) sondern das Register RBP (siehe 2.4) zeigt an eine andere Stelle. Sobald wieder neue Frames hinzukommen, die auf den selben Adressen des bereits abgearbeiteten Frames liegen, wird das bereits gelesene und aus dem „Gedächtnis“ der Register und der CPU gelöschte Frame überschrieben, sodass es endgültig gelöscht wird.

4.1 Stack Overflow

Bei einem Stack Overflow reicht der Arbeitsspeicher eines für den Input bestimmten Buffers nicht mehr aus und das Programm schreibt über den reservierten Arbeitsspei-

cherbereich hinaus Daten auf den Stack. Dadurch kann es dazu kommen, dass wichtige Daten, Methoden oder Befehle, die auf dem Stack gespeichert sind, verändert werden, die Programmausführung umgelenkt wird oder dass das Programm ganz abstürzt. Mit böswilligen Absichten kann man diese Lücke in der Computersicherheit dann zum Beispiel dazu ausnutzen, eigene Methoden auszuführen oder Zugriff zu einem System zu erlangen (siehe 4.3). [20, 24]

4.2 Selbstgeschriebenes Beispielprogramm

Das folgende selbstgeschriebene C-Programm ist ein Beispiel für einen simplen Stack Overflow. In der Main-Methode des Programms werden eine Integer-Variable „user“ und ein char-Array „buffer“ mit 64 Zeichen deklariert. Danach wird „user“ der Wert 0 zugewiesen und mit der „gets()“-Methode wird das char-Array „buffer“ mit User-Input beschrieben. Danach wird mit einer if-Schleife (dt. Wenn-Bedingung) abgefragt, ob „user“ ungleich 0 ist. Wenn dies zutrifft, enthält man vom Programm die Ausgabe „access“, wenn nicht, erhält man „no access“. Nach der Programmlogik dieses C-Codes kann die Variable „user“ eigentlich nicht ungleich 0 sein, weil man keine Möglichkeit hat, „user“ während der Laufzeit des Programms zu verändern, wodurch der String „access“ niemals ausgegeben werden dürfte.

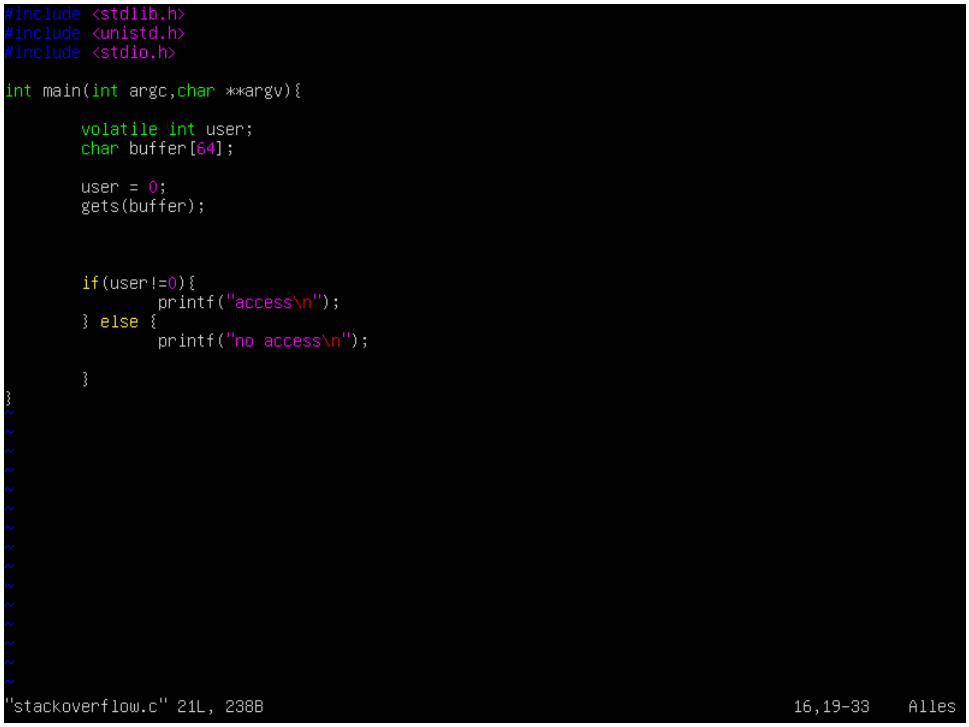


Abb. 7: Beispielprogramm

Wenn die Benutzereingabe beim Ausführen des Programms unter 64 Zeichen hat, (die maximale Größe von „buffer“,) reagiert das Programm wie erwartet.

Abb. 8: Ausgabe bei unter 64 Zeichen

Wird die Benutzereingabe jedoch größer als 64 Zeichern, wird „access“ ausgegeben, was bedeutet, dass sich der Inhalt von „user“ geändert haben muss.

Abb. 9: Ausgabe bei über 64 Zeichen

Im Assemblercode erkennt man, dass der Inhalt von „user“ an einem -0x4 Offset (also der Abstand, der hexadezimal angegeben wird, z.b 0x1B = 16 + 11 = 27) im RBP-Rgister gespeichert wird.

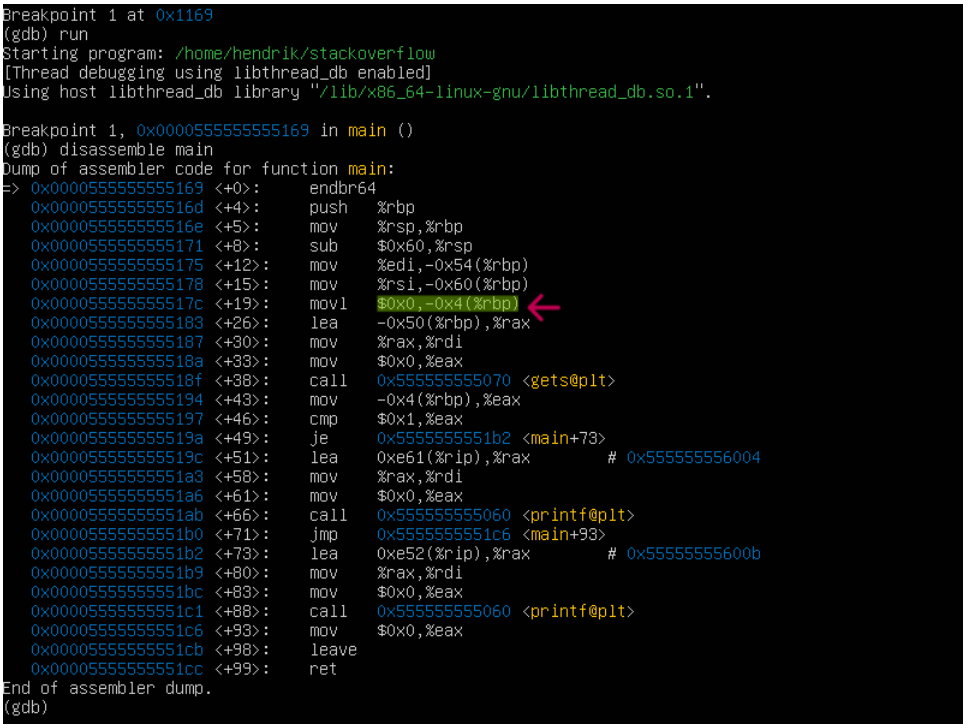


Abb. 10: Disassembly (dt. Zerlegen) von main (dt. sinngemäß Hauptfunktion)

Beim Debuggen (dt. Fehlersuche bzw. -behebung) des Programmes wird klar, dass diese RAM-Adresse bis vor dem Aufruf der „gets()“-Methode auch tatsächlich den Wert 0 besitzt.

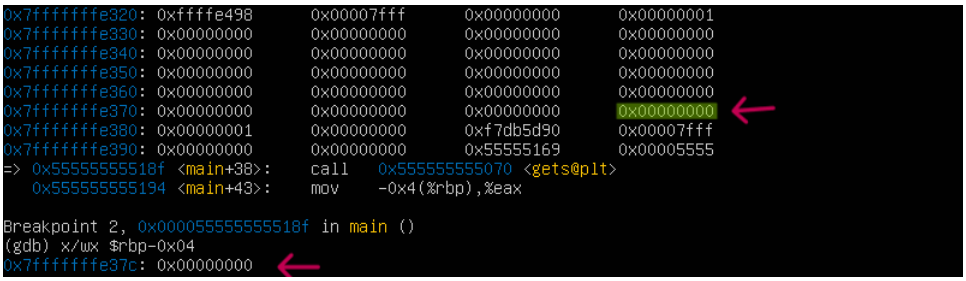


Abb. 11: Stack vor „gets()“

Nachdem die Benutzereingabe von der „gets()“-Methode in „buffer“ gespeichert wurde, kommt es aufgrund des Übersteigens von 64 Zeichen, der maximalen Größe von „buffer“, zu einem Overflow und überzählige Zeichen werden im angrenzenden Bereich gespeichert. Da dieser jedoch eigentlich für „user“ vorgesehen ist, wird der Inhalt von „user“ überschrieben und die if-Bedingung wird erfüllt, sofern der überschüssige Code einen anderen Wert als 0 besitzt, was hier der Fall ist.

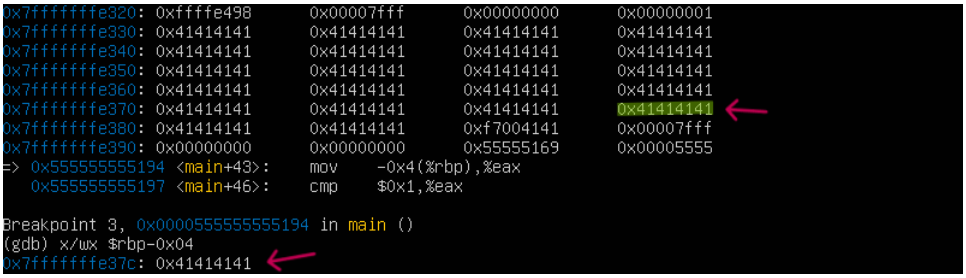


Abb. 12: Stack nach „gets()“

Die „gets()“-Methode, die inzwischen längst veraltet ist, überprüft Eingaben nicht auf Speicherüberlauf im RAM kann ist deswegen extrem gefährlich werden. Die offizielle Linux-Seite empfiehlt deshalb: „Never use gets()“ [13]

Dieses Programm würde auf einem modernen Betriebssystem dennoch keine Probleme verursachen. Damit es hier zu einem Buffer Overflow kommt, müssten Stackguard (siehe 5.5) und die Compileroptimierung (siehe 5.6) deaktiviert werden.

4.3 Stack Overflows ausnutzen

Unabsichtlich die Programmausführung zu verändern (siehe 3.2 und 4.2), ist bei modernen Programmen im Laufe der Zeit extrem unwahrscheinlich geworden. Jedoch werden die Schwachstellen von unsicheren Methoden wie „gets()“ immer noch oft böswillig von Angreifern missbraucht. Diese nutzen den Stack Overflow, um lokale Variablen zu ändern (siehe 4.2) oder um die Return-Adresse so zu überschreiben, dass die CPU zum eigenen, meist schädlichen Code, den der Angreifer auf den Stack geschrieben hat, zurückspringt und diesen ausführt.

Wenn Angreifer ein Computerprogramm attackieren, beginnen sie zuerst damit, alle Informationen über das Programm zu sammeln, die sie finden können. Mit einem Debugger wie gdb (GNU Debugger) für Linux werden Sicherheitsmaßnahmen des Programms, Arbeitsspeicherlayout und Laufzeitverhalten analysiert. Durch sogenanntes Reverse Engineering

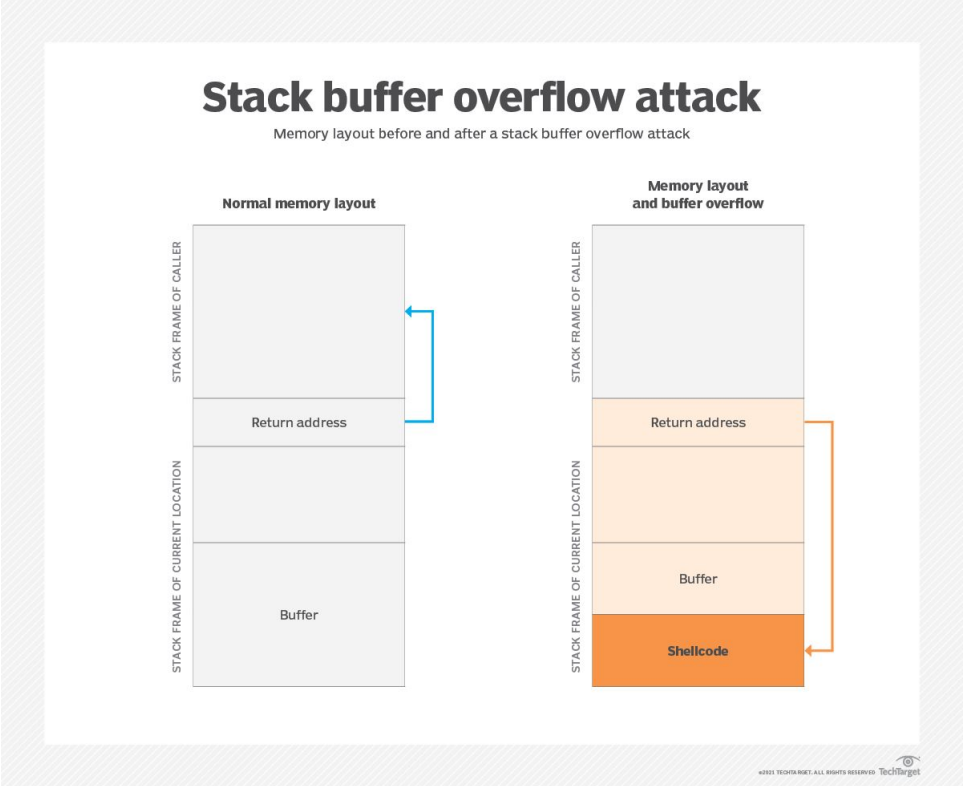


Abb. 13: Ein Stack Overflow Exploit

(dt. Zurückentwickeln bzw. -übersetzen) der Binärdatei kann der Maschinencode in Assemblersprache übersetzt werden, die für den Menschen verständlich ist. Dafür nutzen Angreifer sogenannte Disassembler, wie z.B. IDA (Interactive Disassambler). [20] Aufgrund von zahlreichen Compiler-Optimierungen, die zwar die Ausführungsgeschwindigkeit erhöhen, dafür jedoch den Code so unübersichtlich machen, dass er nur noch extrem schwierig und mit hohem Zeitaufwand zurückübersetzt (reverse engineert) werden kann, ist Reverse Engineering kaum noch lohnenswert. Außerdem wird von manchen Entwicklern (meist von Entwicklern von Schadsoftware) mit zufälligen, nicht verwendeten Codeschnipseln, die in den Code eingefügt werden, ein Reverse Engineering sogar aktiv erschwert, weil der nicht verwendete Code den Disassembler meistens auf falsche Fährten schickt.

Durch ausgiebiges Studieren des Codes versuchen Angreifer also, die Programmlogik zu verstehen und Schwachstellen wie Overflow-anfällige Methoden oder veralteten und umgehbaren Schutz im Code zu entdecken.

Nachdem eine Schwachstelle ausgemacht wurde, im Falle des Stack Overflow eine dafür anfällige C-Methode wie „gets()“ oder „strcpy()“ [36], versuchen die Angreifer im Debugger den Programmstatus zum Methodenaufruf herauszufinden. Dafür werden die Inhalte von

Registern ausgelesen, das Stackframe analysiert und die darauf folgenden Befehle überprüft. Ist eine Adresse, wie ein Pointer, ein Register oder eine Variable gefunden, die überschrieben werden kann, um den Programmablauf im Sinne des Angreifers zu beeinflussen, wird der Offset von Arbeitsspeicheranfang und Zieladresse bestimmt. Dies geschieht beispielsweise, indem man zuerst den Overflow mit Daten durchführt, die einem bestimmten Muster folgen wie „aaaabbbbccccddddeeeeffffgggg“, um danach den Pointer oder die Variable auszulesen.

Anschließend wird eine Exploit-Strategie erstellt, die den Ablauf des Angriffs plant. Meistens besteht diese aus vielen einzelnen Exploits (also einzelne Overflows, Bugs etc.) und verschiedenen Techniken, die zusammen etwas Komplexeres ausführen, wie typischerweise eine Shell zu erzeugen, also eine Schnittstelle zwischen Betriebssystem und Benutzer, in welcher der Benutzer Befehle eingeben kann, die das System dann ausführt. So können sie sich beliebigen Zugriff, meistens Administratorrechte, auf den Rechner verschaffen.

Dann wird ein Exploit-Script vorbereitet, welches Daten und Funktionen enthält, um den Exploit durchzuführen. Bei einem Buffer Overflow wäre das ein Padding, also unwichtige Fülldaten, die den Bereich zwischen Anfang des Buffers und Zieladresse einnehmen und der Payload, der Wert, mit dem die Adresse überschrieben werden soll, die vom Exploit-Script dem Programm übergeben werden. Durch Ändern dieser Adresse, die im optimalen Fall für den Angreifer die Rücksprungadresse darstellen kann, ändert der Hacker den Befehlszyklus und damit auch die Programmausführung, indem er zum Beispiel eine Methode aufruft, die ihm mehr Rechte verleiht.

Dies muss jedoch genau geplant werden, denn wenn der Befehl auch nur um einen einzigen Bit nach links oder rechts verschoben geschrieben wurde, kann das zu Crashs oder Datenverlust führen, da der Stack Pointer nicht mehr an den Beginn des Befehls zeigt und deswegen der Befehl aufgrund des ein- oder ausgeschobenen Bits nicht mehr als dieser erkannt wird.

Ein beliebtes Beispiel für die Ausnutzung eines Overflows ist, dass der Angreifer sich Zugriff auf einen fremden Computer via Internet verschafft, nachdem der Benutzer des angegriffenen Systems besipielsweise eine schädliche Datei oder Software aus dem Internet installiert hat. Hat er es also geschafft, einen Stack Overflow auszunutzen und sich damit Administratorrechte am Computer zu verschaffen, kann er mit sehr guten Programmierkenntnissen im Hintergrund, also für den Benutzer nicht sichtbar, eine Software für das Fernsteuern des gekaperten Rechners aktivieren. Sobald er diesen Zugriff

aktiviert hat, hat er vollen Administratorzugriff auf den Computer und kann ihn nach Belieben fernsteuern.

5 Cybersecurity

Im Laufe der Zeit wurden zahlreiche Konzepte zum Schutz vor Buffer Overflow Exploits entwickelt. Dabei gibt es zwei generelle Ansätze, sich dem Problem zu nähern: Die Abwehr auf Betriebssystemsebene und die Compiler Protection (dt. Kompilierschutz). [10, 16, 24]

5.1 Betriebssystemebene

Um ein Programm vor Sicherheitslücken zu schützen, kann das Betriebssystem nur die äußeren Umstände ändern, in welchen das Programm ausführt werden soll, weil es ja bereits kompiliert wurde und deshalb nicht mehr verändert werden kann. Aus diesem Grund setzen die Gegenmaßnahmen allesamt an der Optimierung der Architekturbestandteile und Verbesserung der Sicherheitsmaßnahmen, wie zum Beispiel der CPU, den Registern oder dem RAM an. [7]

5.2 DEP und NX-Bit

Die Data-Execution-Prevention, kurz DEP und das No-Excecution-Bit, kurz NX-Bit, sind aus einer Weiterentwicklung des Protected Mode [37], der erstmals mit der Intel-80286-CPU eingeführt wurde, entstanden. Der Protected Mode markiert Speicherbereiche im RAM zwar als les- und schreibbar, aber als nicht ausführbar, wodurch es erheblich erschwert wird, schädlichen Code auszuführen. Für neuere Prozessoren wurden als Antwort auf das Problem der zahlreichen Buffer Overflows DEP in Windows und NX-Bit in Linux eingeführt.

Die DEP und das NX-Bit markieren (wie der Protected Mode auch) wichtige Bereiche des Arbeitsspeichers, wie den Stack oder den Heap, die zur Datenspeicherung vorbehalten sind, als nicht ausführbar. Damit der Prozessor erkennt, ob ein Speicherbereich ausführbar ist oder nicht, wird ein einzelnes Boolean-Statusbit (Ja/Nein-Bit) in der Seitentabelle des Prozesses gespeichert. Wenn dann versucht wird, etwas auf dem Stack auszuführen und dadurch eine Zugriffsverletzung erzeugt wird, weil dieser Teil als nicht ausführbar markiert ist, behandelt das Betriebssystem diesen Fehler zum Beispiel durch einen Absturz und anschließendem Neustart des fehlerhaften Programms.

```c
#include <stdio.h>

int main(){
char buffer[64];
gets(buffer);
```

Abb. 14: C-Programm mit gefährlicher „gets()“-Methode

```
hendrik@hendrikspc:~$ xxd exp
00000000: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000010: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000020: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000030: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000040: 9090 9090 20d0 ffff 10d5 ffff 9090 9090  .... ..........
00000050: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000060: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000070: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000080: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000090: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000d0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000e0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000f0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000100: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000110: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000120: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000130: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000140: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000150: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000160: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000170: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000180: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000190: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001d0: 9090 9090 9090 9090 9090 9090 31c0 5068  ............1.Ph
000001e0: 2f2f 7368 682f 6269 6e89 e389 c189 c2b0  //shh/bin.......
000001f0: 0bcd 8031 c040 cd80                      ...1.@..
```

Abb. 15: exp-File

Abbildung 14 zeigt ein Input exp File für das C-Programm, um auf dem Stack den Return-Pointer zu überschreiben und eine Shell mit „shh/bin“ aufzurufen.

```
(gdb) r < exp
Starting program: /home/hendrik/return < exp
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
process 1082 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 1082) exited normally]
```

Abb. 16: Output mit deaktivierten DEP und NX-Bit

Wenn DEP und NX-Bit deaktiviert sind, wird der Prozessor nicht daran gehindert, den Code, der durch das Input exp File auf dem Stack hinterlegt wurde, auszuführen, und die Shell wird gestartet.

Es kommt zu einem sogenanntem Segmentation Fault (dt. Segmentierungsfehler), da die CPU als nicht ausführbar markierten Code auf dem Stack ausführen will, wodurch das Programm vom Betriebssystem abgebrochen wird.

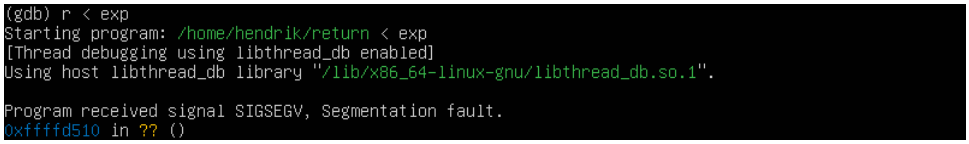


Abb. 17: Output mit aktivierten DEP und NX-Bit

5.3 ASLR

Die Adress Space Layout Randomization (kurz ASLR) ist eine Technik, mit welcher man es Angreifern deutlich erschwert, die Adressen von Funktionen und Strukturen herauszufinden. Die Speicheradressen und Struktur wichtiger Programmteile, wie dem Stack, dem Heap oder verwandten Bibiliotheken, wie die bei Angreifern beliebte „libc“-Datei, werden dafür bei jedem Programmaufruf verändert, sodass ein Angreifer Adressen von früheren Programmausführungen nicht mehr verwenden kann. In einem 32-Bit-System („reines“ x86) ändern sich dabei jeweils die 2 letzten Bytes der Speicheradresse, was 2^{16} verschiedenen Kombinationen entspricht. Die richtigen 2 Bytes herauszufinden, ist mit Brute Force in einem real umsetzbaren Zeitraum durchaus möglich. Bei 64-Bit-Systemen (x86-64), wie sie heutzutage schon lange Standard sind, ändern sich bei jeder Programmausführung immer 4 Byte der Speicheradressse, was 2^{32} Möglichkeiten entspricht und mit Brute Force nicht mehr in einem realistischem Zeitraum zu bewältigen ist. [4, 6]

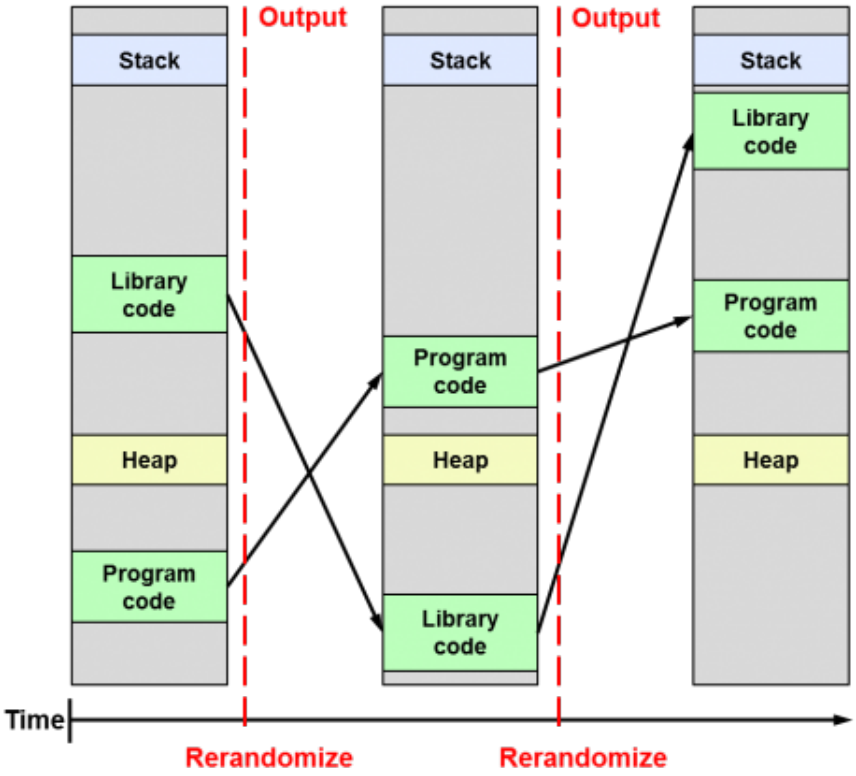


Abb. 18: Konzept von ASLR

5.4 Compiler Level Protection

Der Compiler besitzt im Gegensatz zum Betriebssystem internes Wissen über Arbeitsspeicher und Programmstruktur. Aus diesem Grund hat dieser auch die Möglichkeit, den Aufbau von Programmteilen gezielt zu verändern, ohne mit großen Nachteilen für die Programmausführung rechnen zu müssen. Aus diesem Grund gibt es gegen Binary Exploitations deutlich mehr Sicherheitstechniken auf Compilerlevel als auf Betriebsystemlevel. Als Gegenmaßnahme für den Stack Overflow gibt es jedoch nur Stackguard.

5.5 Stackguard

Stackguard benutzt sogenannte Stack Canaries, geheime Werte, die vor der Rücksprungadresse auf dem Stack gespeichert werden. Sie ändern sich mit jedem Neustart des Programms. Am Ende jeder Funktion werden die Canaries überprüft. Bei einer Änderung des Wertes wird die „abort“-Methode aufgerufen, die das Abbruchsignal SIGABRT (engl. kurz für Signal abort) sendet und das Programm direkt abbricht. In 32-Bit-Systemen sind die Canaries 4 Byte groß, während sie in 64-Bit Systemen meistens 8 Byte groß sind.

Wie bei ASLR ist es in 32-Bit-Systemen noch möglich, die Canaries mir Brute Force herauszufinden, in 64-Bit-Systemen jedoch nicht. Bei moderneren Implementierungen werden Canaries nicht nur vor der Rücksprungadresse, sondern auch vor weiteren kritischen Elementen platziert.

```
Starting program: /home/hendrik/return
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
a
[Inferior 1 (process 15999) exited normally]
(gdb) disassemble main
Dump of assembler code for function main:
   0x565561ad <+0>:     push   %ebp
   0x565561ae <+1>:     mov    %esp,%ebp
   0x565561b0 <+3>:     push   %ebx
   0x565561b1 <+4>:     sub    $0x40,%esp
   0x565561b4 <+7>:     call   0x565561fb <__x86.get_pc_thunk.ax>
   0x565561b9 <+12>:    add    $0x2e1b,%eax
   0x565561be <+17>:    lea    -0x44(%ebp),%edx
   0x565561c1 <+20>:    push   %edx
   0x565561c2 <+21>:    mov    %eax,%ebx
   0x565561c4 <+23>:    call   0x56556050 <gets@plt>
   0x565561c9 <+28>:    add    $0x4,%esp
   0x565561cc <+31>:    mov    $0x0,%eax
   0x565561d1 <+36>:    mov    -0x4(%ebp),%ebx
   0x565561d4 <+39>:    leave
   0x565561d5 <+40>:    ret
End of assembler dump.
```

Abb. 19: Disassembly von main (siehe 4.2) mit deaktiviertem Stackguard

Wenn Stack Canaries aktiviert sind, besitzt die main-Methode noch zusätzliche 4 Zeilen Assemblercode, in denen in Zeile 53 der Wert der Canaries mit „jump if equal“ überprüft wird und bei einer Änderung des Canaries die Methode `__stack_chk_fail_local` aufgerufen wird.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x565561bd <+0>:     push   %ebp
   0x565561be <+1>:     mov    %esp,%ebp
   0x565561c0 <+3>:     push   %ebx
   0x565561c1 <+4>:     sub    $0x44,%esp
   0x565561c4 <+7>:     call   0x56556228 <__x86.get_pc_thunk.ax>
   0x565561c9 <+12>:    add    $0x2e07,%eax
   0x565561ce <+17>:    mov    %gs:0x14,%edx
   0x565561d5 <+24>:    mov    %edx,-0x8(%ebp)
   0x565561d8 <+27>:    xor    %edx,%edx
   0x565561da <+29>:    lea    -0x48(%ebp),%edx
   0x565561dd <+32>:    push   %edx
   0x565561de <+33>:    mov    %eax,%ebx
   0x565561e0 <+35>:    call   0x56556050 <gets@plt>
   0x565561e5 <+40>:    add    $0x4,%esp
   0x565561e8 <+43>:    mov    $0x0,%eax
   0x565561ed <+48>:    mov    -0x8(%ebp),%edx
   0x565561f0 <+51>:    sub    %gs:0x14,%edx
   0x565561f7 <+58>:    je     0x565561fe <main+65>
   0x565561f9 <+60>:    call   0x56556230 <__stack_chk_fail_local>
   0x565561fe <+65>:    mov    -0x4(%ebp),%ebx
   0x56556201 <+68>:    leave
   0x56556202 <+69>:    ret
End of assembler dump.
```

Abb. 20: Disassembly von main mit aktiviertem Stackguard

Beim Ausführen des Codes mit dem selben Input exp-File wie in 5.2 kommt es wie erwartet zum Programmabbruch durch SIGABRT.

Abb. 21: Fehlermeldung SIGABRT durch Stackguard

5.6 Compileroptimierung

Compileroptimierungen sind Methoden, die vom Compiler verwendet werden, um den Code effizienter auszuführen. Ziel ist es, dabei vor allem den benötigten verwendeten Arbeitsspeicher zu minimieren und die Laufzeit so gering wie möglich zu halten. Indirekt kann das Programm dadurch auch Schwachstellen für Buffer Overflows beseitigen. Der Compiler entfernt toten Code, das bedeutet, er löscht Code, der von der Programmlogik her nie ausführbar wäre, da nicht auf ihn verwiesen wird (siehe 4.2). Außerdem kann er kleinere Unterfunktionen in der aufrufenden Funktion einbauen und so verhindern, das Call-Adressen überschrieben werden. Zusätzlich kann der Compiler lokale Variablen in Register verschieben, sodass diese nicht auf dem Stack gespeichert werden müssen. Indem er das Padding von Frames kleiner macht, hat ein Angreifer auch weniger Platz für schädlichen Code, den er auf den Stack schreiben kann. [1, 28, 31]

6 Cybersecurity umgehen

6.1 Stackguard umgehen

Wie bei ASLR kann ein Angreifer auch über Information Leaks die Werte der Canaries erhalten und die Werte dann seinem Payload (dt. sinngemäß die Gesamtmasse an Daten, die später mit einem Mal in den RAM zum Ausführen geschoben werden) hinzufügen.

Anstatt den Return-Pointer zu überschreiben, ist es auch möglich, andere Pointer zu überschreiben, die auch auf dem Stack liegen, ohne die Canaries zu verändern. Wenn ein Function-Call die Adresse eines Strings benutzt, die auf dem Stack gespeichert ist, ist es möglich, diese Adresse mit der eines anderen Strings zu überschreiben, um die Funktionsausführung zu verändern. Alternativ können auch die Adressen von aufgerufenen Funktionen, wenn diese auf dem Stack gespeichert sind, überschrieben werden.

Außerdem ist es möglich, einen sogenannten Global Offset Table Overwrite durchzuführen, um den Eintrag der `stack_chk_fail`-Funktion in der GOT (kurz für Global Offset Table, dt. sinngemäß eine Tabelle mit Offsets von allen Dateien in RAM) so zu ändern, dass anstatt von `stack_chk_fail` eine andere Funktion aufgerufen wird, die das Programm nicht

abstürzen lässt. Dazu kann zum Beispiel ein Format String Overflow benutzt werden oder auch die „write()“ Function der libc-Datei. [29]

Wenn ein Program mit Threads arbeitet, werden die Stacks der Unterthreads unter dem Arbeitsspeicherbereich des Hauptthreads gespeichert. Zudem wird das Canary des Hauptthreads, auch Master Canary genannt, nur kopiert (Thread Canary) und es wird kein neues generiert. Die Unterthreads vergleichen beim Verlassen einer Methode dann nur noch ihr Threadcanary mit dem des Hauptthreads. Wenn ein Angreifer dann auf dem Stack eines der Unterthreads einen Stack Overflow durchführt, mit dem er gleichzeitig das Threadcanary und das Mastercanary überschreibt, wird der Stackguard die Änderung nicht erkennen und der Angreifer kann zum Beispiel mit einer ROP-Chain und ret2libc eine Shell erzeugen. [30]

6.2 ASLR umgehen: Information Leaks

Information Leaks sind Informationen über Speicheradressen des Programmes, die ein Angreifer aus anderen Quellen bekommen hat, wie zum Beispiel einem anderem Typ des Buffer Overflows. Da die Offsets innerhalb der Programmteile immer gleich sind, kann somit die Adresse von anderen kritischen Funktionen oder Daten berechnet werden und ASLR umgangen werden.

6.3 DEP und NX-Bit umgehen: ret2libc

Um DEP und das NX-Bit zu umgehen, kann man auch die Methoden der „libc“-Datei benutzen (meistens die Systemmethode), die eine Shell liefern kann. Im Gegensatz zum ROP führt man dabei aber meistens nur eine Funktion aus, anstatt sich aus vielen Gadgets den Code selbst zusammenzubauen. Wenn mehrere Techniken, um ASLR oder DEP und NX-Bit umgehen zu können, kombiniert werden, kann der Exploit deutlich gefährlicher werden: Ein Angreifer könnte beispielweise mit Hilfe eines Information Leaks die Basisadresse der libc-Datei erhalten und damit den Offset zur Systemmethode berechnen. Dann würde er mit Kontrolle über den Stack ein passendes Gadget auswählen wie „pop rdi, ret 0“. Damit umgeht man ein Argument und kann „/bin/sh“ aufrufen, um eine Shell auszuführen.

6.4 ASLR, DEP und NX-Bit umgehen: ROP

Mit dem sogenanntem Return-Oriented-Programming (dt. rücksprungorientiertes Programmieren), kurz ROP, kann man mit Kontrolle über den Stack kleine, schon existierende

Codeschnipsel, auch Gadgets genannt, hintereinander laufen lassen, um komplexere Methoden oder Programme auszuführen. Diese Codeschnipsel sind Teile von Funktionen oder gelinkten (aufeinander verweisende) Bibliotheken des Programmes und bestehen aus nur wenigen Zeilen Assemblercode. Wenn der Angreifer Informationen über die Speicheradresse einer Funktion oder einer Bibliothek besitzt, kann er den Offset zu dem Teil des Codes, also das Gadget, das er verwenden will, berechnen und ausführen, indem er mithilfe eines Buffer Overflows einen Function Call manipuliert und damit eine Rücksprungadresse überschreibt. Der Angreifer wählt dabei immer Gadgets, die mit einer ret- oder call-Anweisung enden, welche er wieder überschreiben kann, um so beliebig viele Gadgets aneinanderzureihen. Dadurch muss kein Code auf dem Stack ausgeführt werden, denn der Exploit läuft nur im Speicherbereich ab, der für die Programmausführung vorgesehen ist.

```
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _num$[ebp]
    imul   eax, DWORD PTR _num$[ebp]
→   pop    ebp
    ret    0
```

Abb. 22: Das Gadget „pop ebp, ret“ befindet sich an der Speicheradresse 0xffff9117

Wenn ein Angreifer dieses Gadget benutzen will, um zum Beispiel ebp auf 0x00000001 zu setzten, kann er den Return Pointer mit der Adresse des Gadgets überschreiben.

	ret	
%eip: x %esp: 0xffff9404	0xffff940c	0xffff91d8
	0xffff9408	0x00000001
	0xffff9404	0xffff9117
	0xffff9400	0x41414141
	0xffff93fb	0x41414141
	0xffff93f7	0x41414141
	0xffff93f3	0x41414141

Abb. 23: Status des Stacks, bevor ret aufgerufen wird

Da mit „pop esp“ ebp den Wert von esp annimmt, überschreibt der Angreifer das nächste Word (Dateneinheit der Größe 4 Byte in 32-Bit-Systemen) mit dem gewünschten Wert, hier 0x00000001.

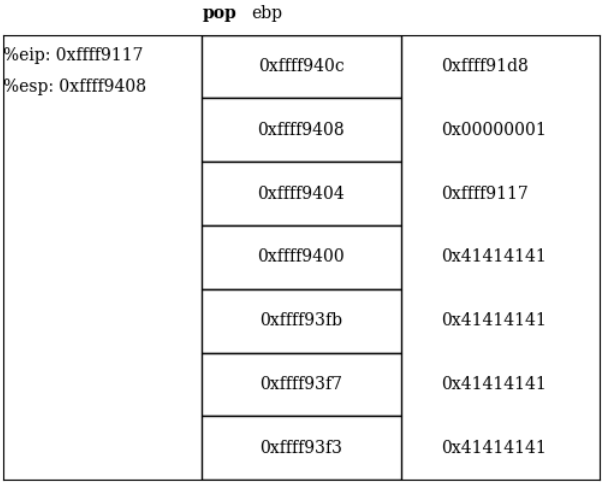


Abb. 24: Status des Stacks, bevor pop ebp aufgerufen wird

Nun wird wieder „ret“ aufgerufen und der Angreifer kann zu einem anderem Gadget springen, hier an der Adresse 0xffff91d8.

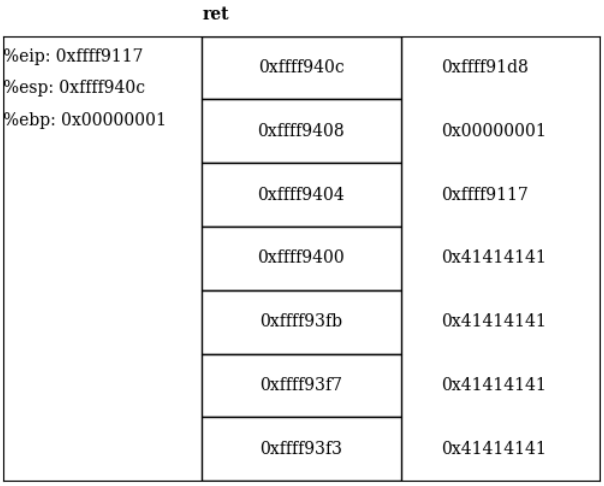


Abb. 25: Status des Stacks, bevor „ret“ ein weiteres Mal aufgerufen wird

Auf diese Weise kann ein Angreifer beliebig viele Gadgets aneinanderreihen. Dies wird auch als ROP-Chain (dt. Kette, Aneinanderreihung) bezeichnet. [6, 22, 26]

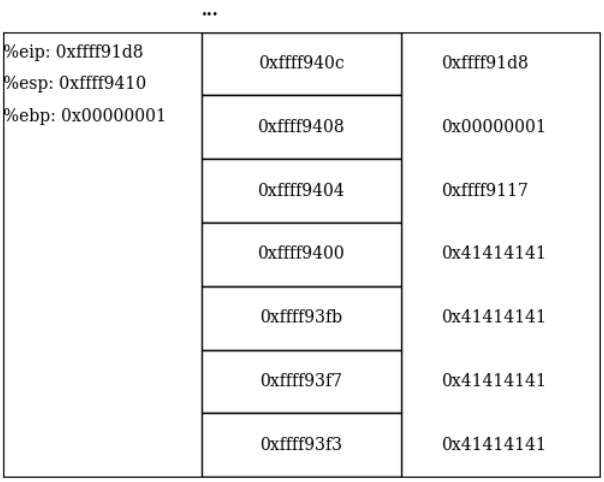


Abb. 26: Status des Stacks, bevor das nächste Gadget aufgerufen wird

7 Ergebnisse

7.1 Diskussion

Dieses Projekt behandelt vor allem den Stack Overflow. Andere Arten des Buffer Overflows wurden in der Literaturrecherche und der Dokumentation nicht beachtet. Da Computersysteme und Architekturen jedoch immer sicherer werden, müssen Angreifer auch immer komplexere Angriffe durchführen, die aus Kombinationen verschiedener Exploits und unterschiedlichen Arten des Buffer Overflow bestehen. Beim „CrowdStrike Disaster“ am 19.07.2024, der größten IT-Katatrophe jemals, bei der weltweit zahlreiche Endgeräte in Flughäfen, Banken und Medien lahmgelegt wurden [35], war zu erkennen, welche Folgen kleinste Fehler nach sich ziehen können. Ein Buffer Overread (ähnlich dem Buffer Overflow) in einem fehlerhaften Update für Windows-Computer führte zum Absturz dieser Systeme. Bei einem Buffer Overread werden, gegensätzlich zum Buffer Overflow, Daten über den Buffer hinaus ausgelesen, was zu Datenleaks oder unvorhergesehenem Verhalten führen kann. Für eine moderne Cybersecurity ist es notwendig, gegen das komplette Spektrum von Angriffen abgesichert zu sein und sich nicht nur auf bestimmte Sicherheitsgebiete zu beschränken.

DEP, NX-Bit, ASLR und Stackguard sind außerdem nicht die einzigen Maßnahmen gegen den Stack Overflow, jedoch die einzigen, die weiträumig verbreitet und relevant sind. Für eine umfangreichere Arbeit müsste dazu auch Recherche zu Sicherheitsmaßnahmen wie Pointer Encryption oder SEH betrieben werden.

7.2 Rückblick

Wir haben gelernt, dass der RAM nicht so einfach aufgebaut ist, wie wir es zum besseren Verständnis in 3.1.2 dargestellt haben, dass ein Buffer Overflow in so vielen Weisen stattfinden kann, dass es unmöglich ist, für immer alle Overflows mit Gegenmitteln zu verhindern und dass der Stack aufgrund der Wichtigkeit und „Macht“ der Befehle im Computer ein besonders beliebtes Ziel bei Angreifern war und immer noch ist, wofür wir mit dem selbst erstellten C-Programm ein tieferes Verständnis bekamen. Außerdem arbeiteten wir uns in das Thema der Binary Security ein, die mit Maßnahmen gegen alle Arten von Buffer Overflows versucht, Computer sicherer zu machen. Beispiele sind das NX-Bit, das verhindert, dass die Teile des Arbeitsspeichers, die nur Daten enthalten, nicht ausgeführt werden können oder die Stack Canaries, die bei Korrumption des betroffenen Arbeitsspeichers das gerade aktive Programm zum Schutz sofort abbrechen.

7.3 Ausblick

In Zukunft möchten wir uns zusätzlich mit anderen Arten von Buffer Overflows befassen, wie zum Beispiel dem Heap, Integer und Format String Overflow, der vor allem kombiniert mit dem Stack Overflow sehr häufig ausgenutzt wurde und immer noch wird. Auch weitere Gegenmaßnehmen wie Pointer Encryption oder SEH werden in der Cybersecurity verwendet, zwar nicht so häufig, aber dennoch oft genug, um sicherheitstechnich eine hohe Relevanz zu besitzen, also werden wir uns auch mit ihnen befassen.

Als praktisches Anwendungsbeispiel möchten wir uns am Beispiel des Nintendo 3DS anschauen, wie Sicherheitslücken von Spielkonsolen ausgenutzt werden. Viele Sicherheitslücken des 3DS wurden im Laufe der Zeit auf verschiedenste Art und Weise ausgenutzt, aber der erste Exploit bestand daraus, das Level-Sharing-Feature (dt. Teilen von Levels) des Spiels Cubic Ninja auszunutzen, bei dem, um ein Level zu empfangen, ein QR-Code gescannt wurde. Wenn man nun selbst einen QR-Code erstellt, kann man ihn so umprogrammieren, dass er jeden Code ausführt, den man will, da der Codeaktivator keine Sicherheitsmaßnahmen besitzt, die das verhindern können. So kann man jeden unsignierten Code ausführen, der bei korrekter Programmierung dazu führt, dass man Homebrew (selbst erstellte Spiele und Anwendungen) ausführen kann.

7.4 Danksagung

Wir bedanken uns herzlich bei allen, die uns dieses Projekt ermöglicht haben.

Wir danken Dr. Hans-Werner und Josephine Hector dafür, dass sie das Hector-Seminar gestiftet und uns dadurch über die Jahre vielfältige Einblicke in das wissenschaftliche Arbeiten ermöglicht haben.

Ein besonderer Dank geht an Markus Götzl, der uns während des Projekts betreut hat und Antworten auf alle unsere Fragen hatte. Für seine Ideengebung, Hilfsbereitschaft und Engagement bedanken wir uns herzlich.

Zu guter Letzt gilt unser Dank allen Kursleiterinnen und Kursleitern, die uns über die Jahre im Hector-Seminar begleitet und unterstützt haben und uns eine spaßige, interessante und lehrreiche Zeit im Hector-Seminar ermöglichten. Besonders hervorheben möchten wir dabei Thomas Knecht, der nicht nur den Kurs KA18 über fantastische 6 Jahre geleitet, sondern auch uns für die Dauer dieses Projekts betreut und uns immer zur Seite gestanden hat.

7.5 Selbstständigkeitserklärung

Hiermit versichern wir, dass diese Arbeit unter der Beratung und Unterstützung von Markus Götzl und unserem Kursleiter Thomas Knecht selbstständig verfasst wurde und keine anderen als die angegebenen Quellen verwendet wurden.

Emilio Craff Castillo

Hendrik Dasselaar

8 Anhang

8.1 Internetquellen

[1] https://medium.com/@ofriouzan/part-2-compiler-level-security-mechanisms-gcc-d01246b8d157 (letzter Zugriff am 03.07.2024)

[2] https://visualgdb.com/documentation/embedded/stackheap/ (letzter Zugriff am 03.07.2024)

[3] https://www.computerweekly.com/de/definition/Buffer-Overflow (letzter Zugriff am 04.07.2024)

[4] https://web.archive.org/web/20230814232415/https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf (letzter Zugriff am 29.07.2024)

[5] https://linz.coderdojo.net/uebungsanleitungen/programmieren/sonstiges/assembler-hello-world/ (letzter Zugriff am 02.07.2024)

[6] https://ctf101.org/binary-exploitation/overview/ (letzter Zugriff am 29.07.2024)

[7] https://infocon.org/cons/Black%20Hat/Black%20Hat%20USA/Black%20Hat%20USA%202007/presentations/Moyer/Extras/bh-us-04-silberman-paper.pdf (letzter Zugriff am 03.07.2024)

[8] http://slowinska.asia/papers/dowser_eurosec13.pdf (letzter Zugriff am 21.07.2024)

[9] https://www.h-ka.de/ (letzter Zugriff am 23.05.2024)

[10] https://www.imperva.com/learn/application-security/buffer-overflow/

[11] https://koreascience.kr/article/JAKO202108038346114.pdf (letzter Zugriff am 03.07.2024)

[12] https://arxiv.org/abs/1807.03757 (letzter Zugriff am 03.07.2024)

[13] https://www.heise.de/tipps-tricks/ (letzter Zugriff am 03.07.2024)

[14] https://www.ajrsp.com/en/Archive/issue-19/The%20Buffer%20Overflow%20Attack.pdf (letzter Zugriff am 31.07.2024)

[15] https://www.careerserviceportal.kit.edu/download/?file=attack_emulation_with_

caldera_110651.pdf&module=uploadfiles&module2=mod_job_exchange&saveas=Attack_Emulation_with_Caldera.pdf (letzter Zugriff am 21.07.2024)

[16] https://ctf101.org/binary-exploitation/what-is-binary-security/(letzter Zugriff am 01.08.2024)

[17] https://www.prosec-networks.com/blog/buffer-overflow-angriff/ (letzter Zugriff am 02.07.2024)

[18] https://www.redlings.com/de/ratgeber/buffer-overflow (letzter Zugriff am 29.07.2024)

[19] https://www.heise.de/en/news/Ever-faster-DDR6-LPDDR6-GDDR7-HBM4-and-PCIe-7-0-in-the-works-9762099.html (letzter Zugriff am 31.07.2024)

[20] https://ctf101.org/reverse-engineering/what-are-disassemblers/ (letzter Zugriff am 31.07.2024)

[21] https://www.heise.de/news/Top-25-der-Sicherheitsluecken-Buffer-Overflows-als-groesste-Gefahrenquelle-6148053.html (letzter Zugriff am 31.07.2024)

[22] https://www.fortinet.com/resources/cyberglossary/buffer-overflow (letzter Zugriff am 03.07.2024)

[23] https://owasp.org/www-community/attacks/Buffer_overflow_attack (letzter Zugriff am 04.07.2024)

[24] https://www.sciencedirect.com/science/article/pii/S1877050919316527 (letzter Zugriff am 04.07.2024)

[25] https://archives.scovetta.com/pub/InfoSec/Analysis%20of%20Buffer%20Overflow%20Attacks.pdf(letzter Zugriff am 04.07.2024)

[26] https://blog.ordix.de/wenn-der-puffer-voll-ist-was-ist-eigentlich-ein-buffer-overflow (letzter Zugriff am 29.07.2024)

[27] https://www.daniloaz.com/en/differences-between-aslr-kaslr-and-karl/ (letzter Zugriff am 03.07.2024)

[28] https://arushs-notes.gitbook.io/blog/fundamentals/ret2libc-and-rop (letzter Zugriff am 29.07.2024)

[29] https://book.hacktricks.xyz/binary-exploitation/ (letzter Zugriff am 29.07.2024)

[30] https://dl.acm.org/doi/pdf/10.1145/3453483.3454035 (letzter Zugriff am 29.07.2024)

[31] https://blog.pwntools.com/posts/got-overwrite/ (letzter Zugriff am 29.07.2024)

[32] https://7rocky.github.io/en/ctf/htb-challenges/pwn/robot-factory/#canaries-and-threads (letzter Zugriff am 03.07.2024)

[33] https://www.geeksforgeeks.org/code-optimization-in-compiler-design/ (letzter Zugriff am 03.07.2024)

[34] https://cdn.daniloaz.com/wp-content/uploads/2017/07/ASLR-concept.png (letzter Zugriff am 03.07.2024)

[35] https://www.crowdstrike.com/falcon-content-update-remediation-and-guidance-hub/ (letzter Zugriff am 03.07.2024)

[36] https://www.ibm.com/docs/en/i/7.3?topic=extensions-standard-c-library-functions-table-by-name (letzter Zugriff am 03.09.2024)

[37] https://www.techtarget.com/searchstorage/definition/protected-mode (letzter Zugriff am 08.09.2024)