Unsichtbare Server mit TCP Stealth

Gut verschlossen

Christian Grothoff, Julian Kirsch

Portscans werden von Geheimdiensten und Kriminellen eingesetzt, um verwundbare Systeme zu finden. Mithilfe von TCP Stealth lassen sich Dienste so absichern, dass sie nur autorisierte Verbindungsanfragen akzeptieren – Portscanner bleiben „ausgesperrt“.

Portscanner wie *nmap* werden weithin eingesetzt, um verwundbare Dienste auf Systemen zu finden. Seit Bekanntwerden des Hacienda-Programms der Five-Eyes-Staaten (siehe Kasten) ist klar, dass Portscans nicht nur vereinzelt durch Hacker, sondern auch im großen Stil durch Geheimdienste zum Einsatz kommen. GCHQ, CSEC, NSA und Co. lassen einem erfolgreichen Portscan dann gegebenenfalls gleich noch einen Angriff mit 0-Day-Exploits folgen, um das Opfer als „Operational Relay Box“ (ORB) in das eigene Agentennetz einzupflegen. Eine Verteidigung gegen einen 0-Day-Exploit ist schwer, aber mithilfe von TCP Stealth lassen sich TCP-Server so absichern, dass sie nur noch autorisierte Verbindungsanfragen akzeptieren. Portscanner werden dadurch „ausgesperrt“ und 0-Day-Attacken selbst von Man-in-the-Middle-Angreifern deutlich erschwert.

Wie TCP Stealth funktioniert

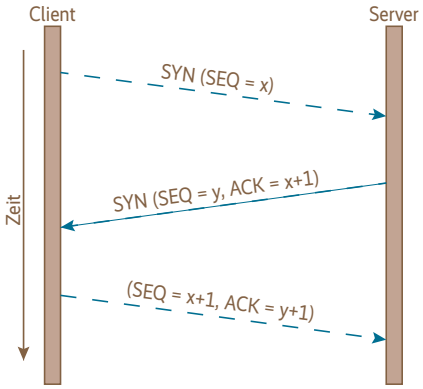
Das sicherheitstechnische Grundübel einer TCP-Anfrage, das Port-Scanning erst ermöglicht, ist die grundsätzliche Antwortbereitschaft des Servers für jede Anfrage. Der antwortet nämlich immer auf die einleitende Verbindungsanfrage, die SYN-Nachricht eines TCP-Drei-Wege-Handschlags, ohne zu wissen, ob es sich bei der Gegenseite überhaupt um einen autorisierten Nutzer handelt. Dadurch ist es möglich, aktive Dienste, die an TCP-Ports gebunden sind, auf beliebigen Rechnern im Internet zu finden. Vor allem bei über das Netzwerk zugänglichen administrativen Schnittstellen wie Telnet, SSH oder SNMP kann das kritisch sein, da ein Angreifer mit einem einfachen SYN-Paket aktive TCP-Server aufgrund ihrer SYN-ACK-Antwort identifizieren kann.

TCP Stealth, eine von den Autoren dieses Artikels entwickelte, mit dem TCP-Standard kompatible Erweiterung, behebt diesen Missstand, indem es in die Verbindungsanfrage des Clients ein Authentifizierungszeichen einbettet. Die initiale Sequenznummer (ISN) im TCP-SYN-Segment wird durch eine in einem Einwegverfahren generierte Zahl ersetzt. Die erlaubt es dem Server, direkt nach Erhalt des Segments zu entscheiden, ob die andere Seite den Dienst verwenden darf. Die Zahl wird aus mehreren Komponenten errechnet, darunter einem beiden Seiten vorliegenden geheimen Schlüssel.

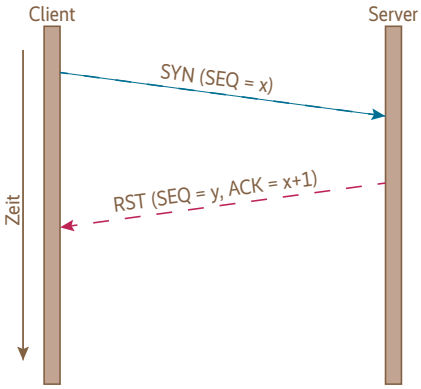
Das Verfahren reiht sich damit in eine lange Schlange sogenannter Port-Knocking-Techniken ein, geht aber einen Schritt weiter. Beim herkömmlichen Port-Knocking autorisiert sich der Anfragende durch ein bestimmtes Klopfzeichen. Kontrolliert allerdings ein Angreifer alle Wege zwischen einem Client und einem Server, kann er dieses Verfahren aushebeln, indem er den Verkehr nach der Autorisierung als „Man in the Middle“ übernimmt.

Zusatzsicherheit per Prüfsumme

Um Letzteres zu verhindern, überträgt TCP Stealth optional zum Authentifizierungszeichen im ersten Segment eine kryptografische Prüfsumme (Hash) der ersten, vom Client nach dem Handschlag gesendeten Datenbytes. In einem Applikationsprotokoll könnten diese ersten Bytes beispielsweise einen öffentlichen



Paketfluss eines erfolgreichen Drei-Wege-Handschlags (Abb. 1)



Paketfluss für einen erfolglosen Versuch an einem geschlossenen Port (Abb. 2)

Schlüssel eines asymmetrischen Kryptosystems darstellen – ein Angreifer zwischen Server und Client wäre dann nicht mehr in der Lage, seinen eigenen Schlüssel nach dem abgefangenen Handschlag im Datenstrom zu substituieren.

TCP Stealth wurde an der Technischen Universität München entwickelt und im August 2014 als Standardisierungsvorschlag an die IETF gesendet (siehe „Alle Links“). Details und weiter reichende Informationen finden sich außerdem in der Masterarbeit „Verbessertes kernelbasiertes Port-Knocking unter Linux“ [1].

Installation unter Linux

Da der Linux-Kernel TCP Stealth (noch?) nicht unterstützt, ist zunächst das Patchen des Kernels erforderlich. Den aktuellen TCP-Stealth-Patch (eine Implementierung namens Knock) gibt es auf der Projekt-Homepage (siehe „Alle Links“). Außerdem müssen die Build Essentials zum Kompilieren des Kernels sowie *libncurses5*, die semigrafische Oberfläche der Kernel-Konfiguration, auf dem System vorhanden sein. Der Patch-Vorgang besteht aus sieben Schritten.

1. Besorgen der Kernel-Quellen. Die gewünschte Version bekommt man auf https://www.kernel.org, sofern man einen Vanilla-Kernel ohne Anpassungen installieren möchte. Ein distributionsspezifischer Quelltext ist je nach Linux-Version auf unterschiedlichen Wegen erhältlich. Bei Distributionen, die den Paketmanager *apt* verwenden, funktioniert

```
sudo apt-get source linux-image-$(uname -r)
```

2. Patchen des Kernels. Im Wurzelverzeichnis der heruntergeladenen Quelltexte eingeben:

```
patch+ -p1 < /pfad/zum/tcp_stealth_patch/ 7
                          tcp_stealth_3.16.diff
```

Die aktuelle Version des Patches unterstützt Linux-Kernel ab Version 3.13.

3. Konfigurieren des neuen Kernels. Die Konfigurationsdatei des aktuell ausgeführten Kernels (*running config*) ist auf verschiedenen Wegen erhältlich.

Debian-Linuxe installieren zu jedem Kernel die verwendete Konfiguration im Verzeichnis */boot*. Diese kann man direkt in den Quelltext-Baum des zu kompilierenden Kernels kopieren:

```
cp /boot/config-$(+uname+ -r) .config
```

Viele andere Distributionen arbeiten mit einem Kernel, der es erlaubt, die Konfiguration zur Laufzeit aus dem *proc*-Dateisystem zu lesen:

```
zcat /proc/config.gz > .config
```

Ubuntu und Debian stellen im Ordner *debian/config/<architektur>* passende Konfigurationsdateien für verschiedene Szenarien bereit, die ebenfalls nur in das Wurzelverzeichnis des Kernel-Quelltextes kopiert werden müssen:

```
cp debian/config/<arch>/config.<flavour> \
.config
```

Sollte keiner der angegebenen Wege zum Erfolg führen, konfiguriert man den Kernel mit den Standardwerten. Der Befehl *make defconfig* generiert eine Standardkonfiguration, die natürlich keine hohen Ansprüche hinsichtlich Performanz und Stabilität erfüllt.

4. Sobald eine Konfigurationsdatei angelegt wurde, setzt der Befehl *yes “” |* *make oldconfig* alle von der Konfiguration nicht abgedeckten Eigenschaften auf Standardwerte.

5. Zum Aktivieren von TCP Stealth in der Konfigurationsdatei wählt man entweder den Punkt „Networking Support > Networking Options > TCP/IP networking > TCP: Stealth TCP socket support“ im *ncurses*-Konfigurationsdialog des Kernels (*make menuconfig*), oder man editiert direkt die *.config*-Datei und setzt dort die Konfigurationsoption *TCP_STEALTH* auf den Wert „y“.

6. Kompiliervorgang starten:

```
make bzImage && make modules
```

Das kann je nach Konfiguration und Maschine zwischen wenigen Minuten und mehreren Stunden dauern. Sollte die kompilierende Maschine mehr als einen

Das Hacienda-Programm der Five-Eyes-Staaten

Als streng geheim klassifizierte Dokumente des britischen Geheimdienstes General Communication Headquarter (GCHQ), die heise online seit August 2014 exklusiv vorliegen, haben gezeigt, dass Geheimdienste das Scannen offener Ports ganzer Länder als Routineverfahren einsetzen. Das GCHQ-Programm mit dem Namen Hacienda bewirbt in einer Präsentation aus dem Jahr 2009 Vollscans für insgesamt 27 Länder, verbunden mit dem Angebot: „Wenn Sie ein weiteres Land brauchen, senden Sie uns eine E-Mail.“ Regeln für die Begutachtung oder Begründungen für eine solche Aktion finden sich in den Dokumenten nicht. Ganze Länder zu scannen ist technisch machbar: Tools wie ZMap erlauben das Ausleuchten des gesamten IPv4-Adressraums in weniger als einer Stunde. Ein derart gezielter, massiver Einsatz der Scans macht tendenziell jeden Server zu einem Ziel staatlicher und anderer Computersaboteure.

Neben aktiven Scans durch Hacienda können die Geheimdienste durch bloßes Beobachten normalen Port-Knocks leicht auf die Schliche kommen. Nachdem ein verwundbarer Dienst entdeckt wurde, leiten die „Five-Eyes“-Geheimdienste ihr „Computer Network Exploitation“(CNE)-Programm zur Übernahme fremder Rechner ein. Solche Rechner stehen anschließend als „Operational Relay Boxes“ (ORBs) für getarnte weitere Angriffe zur Verfügung. Es geht also den Geheimdiensten um mehr als Datensammeln. Es geht darum, tendenziell das ganze Netz für Sabotagezwecke zu kapern.

Listing 1

```
char secret[64] = "This is my magic secret.";
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
setsockopt(sock, TCP_STEALTH, secret, sizeof(secret));
/* Weiter wie gewohnt mit connect(), listen(), ... */
```

Einsatz von TCP Stealth mit Authentisierung der Clientseite, Code ist von Client und Server auszuführen

Listing 2

```
char secret[64] = "This is my magic secret.";
 char payload[42];
 /* Fiktives Beispiel, in welchem der Client als Payload zuerst einen
   * kryptographischen Schluessel sendet. Die Funktion steht exemplarisch
   * fuer eine beliebige Schluesselerzeugung, welche das Ergebnis im Feld
   * payload speichert */
 client_derive_ephemeral_key(payload);
  int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  setsockopt(sock, TCP_STEALTH, secret, sizeof(secret));
  setsockopt(sock, TCP_STEALTH_INTEGRITY, payload, sizeof(payload));
  /* Weiter wie gewohnt mit connect(), ... */
  send(sock, payload, sizeof(payload), 0);
```

Einsatz von TCP Stealth mit Authentifizierung und Datenintegritätsprüfung auf der Clientseite

CPU-Kern besitzen, lässt sich durch den Parameter *–j* für den *make*-Befehl die Zahl der Build Threads erhöhen, was den Vorgang erheblich beschleunigen kann.

7. Den neuen Kernel samt Modulen installieren:

```
sudo make modules_install && sudo make install
```

Nach einem Neustart steht allen Anwendungen die TCP-Stealth-API zur Verfügung.

Der Patch führt drei neue Optionen für den *setsockopt*-Aufruf ein, die in der aktuellen Version die Nummern 26 bis 28 belegen:

```
#define TCP_STEALTH              26
#define TCP_STEALTH_INTEGRITY          27
#define TCP_STEALTH_INTEGRITY_LEN     28
```

Ein Aufruf von *setsockopt* mit *TCP_STEALTH* als Argument aktiviert den Authentifizierungsteil von TCP Stealth. Sowohl Client als auch Server verwenden den TCP_STEALTH-Aufruf, um das gemeinsame Geheimnis an die jeweilige TCP-Stealth-Implementierung zu übergeben.

TCP Stealth in Benutzerapplikationen

Es sei noch einmal betont, dass alle TCP-Stealth-relevanten Aufrufe vor dem Senden der Verbindungsanfrage durch *connect* auf der Client- und vor dem ersten Entgegennehmen von Verbindungen durch *accept* auf der Serverseite durchgeführt werden müssen.

Anschließend kann die oben erwähnte Datenintegritätsprüfung aktiviert werden. Clients erledigen das durch einen *setsockopt*-Aufruf mit dem Argument *TCP_STEALTH_INTEGRITY*, das die Daten spezifiziert, die direkt nach dem Verbindungsaufbau gesendet werden. Wie erwähnt, sollte eine Applikation an dieser Stelle die Daten durch kryptografische Verfahren gegen Veränderung sichern – TCP Stealth verschlüsselt nicht, die Daten sind zwar gegen eine Modifikation durch einen Man-in-the-Middle-Angriff geschützt, allerdings für jeden lesbar. TCP Stealth kann außerdem Wiedereinspielungsattacken (sogenannte Replay-Angriffe) verhindern, sofern sich die Daten in der Integritätsprüfung nicht einfach wiederholen und die Applikation ein geeignetes Verfahren bezüglich der geschützten Daten verwendet.

Wie in Listing 2 zu erkennen ist, übernimmt der Aufruf von *setsockopt* nicht das eigentliche Senden der Daten – vielmehr sorgt er dafür, dass die später zu übertragenden Daten bei der Berechnung der Integritätsprüfsumme berücksichtigt werden.

Auf der Serverseite gibt der Aufruf von *setsockopt* durch das Argument *TCP_STEALTH_INTEGRITY_LEN*>+ an, wie viele Datenbytes beim Verifizieren der Integritätsprüfsumme zu berücksichtigen sind. Für das korrekte Funktionieren von TCP Stealth muss die Anzahl der zu schützenden und zu prüfenden Bytes auf Client- und Serverseite übereinstimmen. Die Authentisierung und Datenintegritätsprüfung auf der Serverseite könnte also aussehen wie in Listing 3 gezeigt.

Listing 3

```
char secret[64] = "This is my magic secret.";
  /* Die Zahl payload_len entspricht der Laenge des payload-Arrays das im
   * setsockopt-Aufruf mit TCP_STEALTH_INTEGRITY auf der Clientseite
   * verwendet wird */
  size_t payload_len = 42;
  int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  setsockopt(sock, TCP_STEALTH, secret, sizeof(secret));
  setsockopt(sock, TCP_STEALTH_INTEGRITY_LEN, payload_len, sizeof(payload_len));
  /* Weiter wie gewohnt mit listen(), accept(), ... */
```

Einsatz von TCP Stealth mit Authentisierung und Datenintegritätsprüfung auf der Serverseite

Listing 4

```
$ # Verbindungsversuch zu einem mit TCP Stealth geschuetzten Service
$ ncat tcpstealth.test.server.com 4242
Ncat: Connection refused.
$ # Setzen der TCP-Stealth-Parameter
$ KNOCK_SECRET="This is my magic secret."
$ KNOCK_INTLEN=13
$ LD_PRELOAD=./libknockify.so
$ # Die Verbindungsanfrage wird nun angenommen
$ ncat tcpstealth.test.server.com 4242
Hello Server.
```

Einbinden der dynamischen Bibliothek libknockify.so durch Modifikation der Variable LD_PRELOAD

Damit sind die drei Aufrufe der TCP-Stealth-API beschrieben. Auf der Projekt-Webseite (siehe „Alle Links“) findet man vollständige, kompilierbare Codebeispiele.

Aktivieren von TCP Stealth ohne Codeänderungen

TCP Stealth lässt sich auch ohne Änderungen im Quelltext nutzen. Hierzu dient die Bibliothek *libknockify*, die ebenfalls von der Projekt-Homepage bezogen werden kann. Im Wesentlichen fungiert sie als Wrapper für die bestehenden Funktionen der Netzwerk-API. Zum Einsatz kommt sie durch Eintragen im *LDPRELOAD*-Pfad des dynamischen Linkers *ld*. So lässt sich das „Schweizer Armeemesser zur Netzwerkanalyse“, *ncat*, so modifizieren, dass es sich mit einem durch TCP Stealth geschützten Service verbinden kann, wie der Code in Listing 4 demonstriert.

Die Bibliothek kann entweder durch Umgebungsvariablen oder durch eine Datei namens *.knockrc* im Home-Verzeichnis des jeweiligen Nutzers konfiguriert werden. Das *KNOCK_SECRET* spezifiziert dabei das TCP-Stealth-Geheimnis, die Variable *KNOCK_INTLEN* gibt an, wie viele Datenbytes die Integritätsprüfung berücksichtigen soll. Eine Länge von 0 deaktiviert die Integritätsprüfung. Genauere Details zum Kompilieren und zur Verwendung der *libknockify*-Bibliothek sowie ausführliche Beispiele zu TCP Stealth finden sich auf der Projekt-Homepage. (js)

Julian Kirsch

schließt gerade seinen Master an der Technischen Universität München ab und wird dort künftig am Lehrstuhl für Computersicherheit von Professor Eckert promovieren. Seine Forschungsinteressen sind unter anderem Reverse Engineering und Spionageabwehr.

Christian Grothoff

ist Maintainer von GNUnet und leitet das Team Décentralisé der französischen Forschungseinrichtung Inria in Rennes.

Literatur

[1] Julian Kirsch. Improved Kernel-Based Port-Knocking in Linux. Master Thesis, August 2014. https://gnunet.org/kirsch2014knock

Alle Links: www.ix.de/ix1410136