



Fakultät Informatik
Institut für Systemarchitektur, Lehrstuhl Rechnernetze

Implementierung einer Testsuite zur Untersuchung von Möglichkeiten für DoS-Angriffe auf DTN- Protokollimplementierungen

Tobias Nöthlich
Florian Richter
Tim Krieg

Projektdokumentation

Betreuer
Dr.-Ing. Marius Feldman

Wintersemester 2020/2021

Inhaltsverzeichnis

1	Einleitung	2
1.1	Motivation	2
1.2	Anforderungen	2
2	Implementation	2
2.1	Shell Skripte	2
2.1.1	Setup	2
2.1.2	Start	2
3	Szenarien	2
3.1	Bundle Flooding	2
3.2	DDoS Flooding	4
3.3	Slowloris	4
4	Erweiterung	7
4.1	Aufbau eines virtuellen Netzwerks mit CORE	7
4.1.1	Start des Common Open Research Emulator	7
4.1.2	Aufbau eines Netzwerks mit CORE	8
4.1.3	Integration von ION	8
4.1.4	nodeSetup.sh	8
4.2	Konfiguration von ION	9
4.3	Visualisierung des Szenarios	9
4.4	bundlecount.sh und bundlewatch.sh	9
4.5	pingvis.sh	9
4.6	othervis.sh	10

1 Einleitung

1.1 Motivation

In Katastrophengebieten oder Verzögerungstolerante Netzwerke (engl. Delay-Tolerant Networking, kurz DTN) [PLATZHALTER] Generelles Intro über Nutzen von DTN in Raumfahrt, Katastrophenhilfe, etc

[PLATZHALTER] Kurz erklären wie das ganze funktioniert

[PLATZHALTER] Überleiten auf Angriffsvektoren → unsere Testsuite kommt ins Spiel.

1.2 Anforderungen

- Anforderungsanalyse

2 Implementation

2.1 Shell Skripte

2.1.1 Setup

2.1.2 Start

3 Szenarien

3.1 Bundle Flooding

Das erste Szenario ist die Simulation eines Bundle Flooding Angriffs auf ein Netzwerkelement, das als Verbindungsstück zwischen einem Satelliten und dem Mission Control Center fungiert. Die für die Simulation genutzte Netzwerktopologie (siehe Abb. [PLATZHALTER]) besteht aus fünf Elementen, welche im Common Open Research Emulator durch den **Router** Knotentypen emuliert werden.

Das Grundprinzip des dem Szenario zugrunde liegenden Denial of Service Angriffs ist das Fluten des Zielsystems mit einer so großen Menge an Bundles, dass es nicht mehr in der Lage ist den legitimen Traffic zuzustellen. Da Delay-Tolerant Networking Systeme (DTN-Systeme) insbesondere für Gebiete, in denen stabile Verbindungen nicht durchgehend möglich sind, konzipiert wurden, gibt es auch in diesem Szenario Verbindungsunterbrechungen zwischen Satellit und dem Mission Control Center. Da durch unterbrochene Verbindungen nicht zustellbare Bundles zunächst auf dem letzten erreichbaren Knoten gesichert werden, muss ein Angreifer lediglich einen Knoten mit instabiler Verbindung attackieren. Es ist allerdings von Nöten, dass der Angreifer Zugriff auf ein in das ION-Netz eingebundenes System hat, da ION die verfügbaren Kontakte aus einer Konfigurationsdatei liest. Es ist also nicht möglich, von ausserhalb eine Verbindung zu einem bestehenden Netzwerk herzustellen.

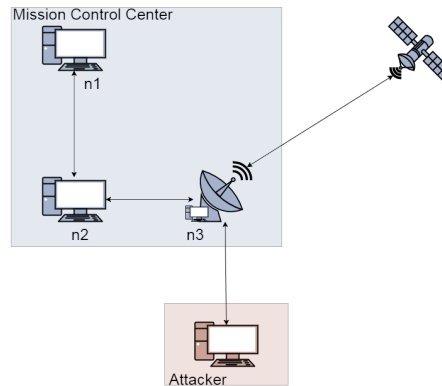


Abbildung 1: Netzwerktopologie des Bundle Flooding Szenarios

Konkret auf unser Szenario bezogen bedeutet dies, dass der *Angreifer* ein System im Netzwerk übernommen hat, und nun eine Verbindung zu *n3* besteht. Er wird nun diesen Knoten mit an den *Satelliten* adressierten Bundles fluten, bis der restliche Netzwerkverkehr zum Erliegen kommt. Dabei reichen 50 Bundles pro Sekunde aus, um in kurzer Zeit die Übertragungskapazität der Leitung zwischen *n3* und dem *Satelliten* so zu verschlechtern, dass nur knapp ein Drittel der ursprünglichen Leistung für den restlichen Traffic zur Verfügung steht. Dies zeigt sich an den Round Trip Times (RTT) der Kommunikation zwischen *n1* und dem *Satelliten*. Hat ein Bundle Ping von *n1* zum *Satelliten* vor dem Start des DoS-Angriffs eine RTT von knapp einer Sekunde, so beträgt diese wenige Sekunden nach Start des DoS-Angriffs bereits über 3 Sekunden. Circa 5 Sekunden nachdem der *Angreifer* beginnt *n3* zu fluten, ist ein Ping von *n1* zum *Satelliten* nicht mehr möglich, da bereits über 100 Bundles vom *Angreifer* auf dem Knoten *n3* auf die Weiterleitung zum *Satelliten* warten.

Ein wichtiger Faktor zum Erfolg dieses Angriffs, ist die von ION gegebene Möglichkeit, die Priorität der vom Bundle Ping ausgesendeten Bundles zu ändern. Durch das Versenden von Bundles mit der höchsten Priorität ist es dem *Angreifer* möglich, die von ihm gesendeten Bundles an den Anfang der Queue zu setzen, die die Reihenfolge der weitergeleiteten Bundles festlegt. Des Weiteren sorgt die instabile Verbindung zwischen *n3* und dem *Satelliten*, welche alle 30 Sekunden für 30 Sekunden unterbrochen wird, für ein rapides Ansteigen der auf *n3* zwischengelagerten Bundles. Die so von nicht legitimen Bundles circa 50 zu 1 dominierte Weiterleitungsqueue kann nicht mehr komplett geleert werden, was nach entsprechender Dauer zum Timeout der von *n1* kommenden Bundles führt.

Zur Visualisierung der Attacke dienen zwei Shell-Skripte, welche die Anzahl der auf jedem Knoten eingelagerten Bundles zeigen, sowie ein Skript, das den Ping von *n1* zum *Satelliten* visualisiert. Dieses Skript nutzt dabei zur besseren Übersicht die von ION zu Verfügung gestellten watch characters. So wird für jedes versendete Bundle einmal das Zeichen 'a' ausgegeben. Erreicht das auf so

ausgelöste Acknowledgement $n1$, wird desweiteren die RTT angezeigt. Die zur Visualisierung benötigten Skripte werden automatisch mit Start des Szenarios ausgeführt.

3.2 DDoS Flooding

Eine Weiterentwicklung des ersten Szenarios ist die Simulation eines von mehreren Angreifern gleichzeitig ausgeführter Bundle Flooding Angriff auf das gleiche Verbindungsstück zwischen einem Satelliten und dem Mission Control Center wie im Bundle Flooding Szenario. Die für die Simulation genutzte Netzwerktopologie (siehe Abb. [PLATZHALTER]) besteht nun aus neun Elementen, welche im Common Open Research Emulator durch den **Router** Knotentypen emuliert werden.

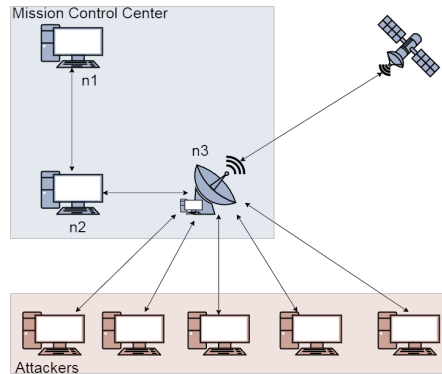


Abbildung 2: Netzwerktopologie des DDoS Szenarios

Das Grundprinzip des Angriffes ist dem im vorherigen Szenarion Verwendeten sehr ähnlich. Hier wird jedoch $n3$ deutlich stärker ausgelastet, da nicht nur ein Angreifer Bundles sendet, sondern fünf.

Tatsächlich ist die Belastung so hoch, dass $n3$ wenige Sekunden nach Start des Angriffes kaum noch Bundles annehmen kann. Es kommt also zu einer Ansammlung von Bundles auf den Angreiferknoten, welche ihre Bundles nicht mehr schnell genug an $n3$ senden können. Ansonsten ist das Ergebnis das Gleiche wie im Bundle Flooding Szenario, wobei der Verbindungsabbruch durch das höhere Bundleaufkommen etwas hier eher auftritt.

3.3 Slowloris

Im Gegensatz zu den bisher diskutierten Szenarios benötigen Application Layer Denial of Service Angriffe kein bereits kompromittiertes System um zu funktionieren. Das in diesem Abschnitt näher betrachtete Szenario setzt dabei einen Slowloris Angriff erfolgreich um. Wir nutzen in dem Szenario slowloris.py (Version 0.2.2; Yaltirakli, G., 2015)[2], ein Python Skript, welches die Attacke auf das Zielsystem ausführt.

Die hier simulierte Situation ist ein Angriff auf ein Katastrophenfrühwarnsystem. Die Netzwerktopologie (siehe Abb. [Platzhalter]) ist die Folgende:

- eine Messstation
- ein Satellit für die Datenübermittlung
- ein Kontrollzentrum
- ein Emergency Broadcast System
- ein Angreifer, der die Verbindung zwischen Messstation und Satellit stört

Dabei sind die Systeme des Frühwarnsystems in Reihe geschaltet und zusätzlich der Angreifer mit dem Satelliten verbunden. Als Transportprotokoll kommt TCP zum Einsatz. Wie auch in den vorherigen Szenarien werden die einzelnen Knoten durch CORE Routerknoten emuliert.

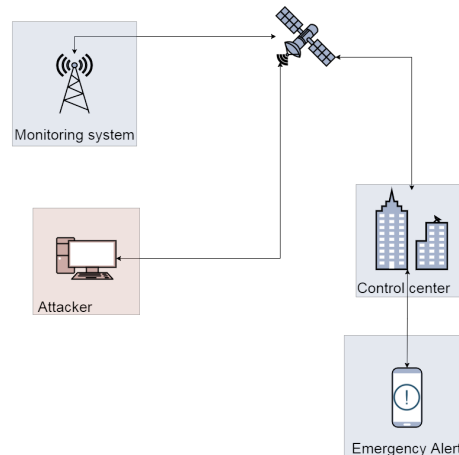


Abbildung 3: Netzwerktopologie des Slowloris Szenarios

Das Ziel des Denial of Service Angriffs ist das Lahmlegen des Zielsystems unter minimaler Verwendung von Netzwerkressourcen. Dies wird erreicht, indem das Angreifersystem möglichst viele Verbindungen zum Zielsystem aufbaut und diese so lange wie möglich offenhält.

In diesem Szenario wird dieser Effekt durch das Ausführen von dem slowloris.py Skript auf dem Knoten, der den Angreifer simuliert, erreicht. Das Skript baut so viele Verbindungen wie möglich zu dem Satelliten auf und versendet nur Teilanfragen. Dies hat den Effekt, dass auf dem Satelliten die Anfragen nie vollständig abgeschlossen werden, und die Verbindungen zum Angreifer bestehen bleiben. Werden auf diese Weise genügend Verbindungen gleichzeitig blockiert, so kann der Satellit von der Messstation eingehende Verbindungsanfragen nicht annehmen und die Verbindung zwischen Messstation und Kontrollzentrum ist unterbrochen. Die Verbindungsunterbrechung tritt bereits den

Bruchteil einer Sekunde nachdem der Angriff gestartet wurde auf und ist somit deutlich schneller als bei dem Bundle Flooding Angriff.

Diese Art Denial of Service Attacke zeichnet sich besonders dadurch aus, dass sie vollkommen vom ION-Netzwerk unabhängig ist, also kein bereits kompromittiertes System innerhalb des Netzwerkes benötigt wird, um erfolgreich die Verbindung zu stören. Ein Angreifer mit ausreichend leistungsstarker Hardware könnte damit sehr einfach sämtlichen Traffic in kritischen Systemen zum Erliegen bringen, sofern diese TCP als Transportprotokoll nutzen.

Die Visualisierung der Auswirkungen des Angriffes in unserer Netzwerksimulation erfolgt wieder durch ein Shell-Skript, welches das Ergebnis des Pings von Messstation zum Kontrollzentrum in einer Konsole ausgibt.

4 Erweiterung

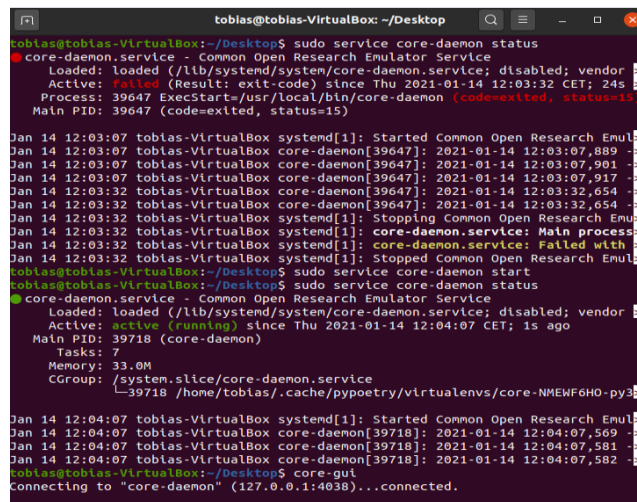
Wir haben bei der Planung und Entwicklung dieses Tools Wert darauf gelegt, dass schnell und einfach neue Szenarien integriert werden können.

4.1 Aufbau eines virtuellen Netzwerks mit CORE

4.1.1 Start des Common Open Research Emulator

CORE (Common Open Research Emulator) ist ein Tool mit welchem man leicht virtuelle Netzwerke aufbauen kann. Als Emulator baut CORE eine Repräsentation eines in Echtzeit laufenden, realen Computernetzwerks auf, anstelle einer Simulation bei der abstrakte Modelle verwendet würden. Diese Emulation kann bei Bedarf auch mit physischen Netzwerken und Routern verbunden werden und bietet eine Umgebung um echte Applikationen und Protokolle zu testen. [1]

Zum Betrieb von CORE ist es wichtig, dass der core-daemon im Hintergrund läuft. Der Status des Daemons lässt sich mittels `sudo service core-daemon status` abfragen. Sollte der daemon nicht gestartet sein, muss `sudo service core-daemon start` ausgeführt werden. Nun kann man mittels `core-gui` die graphische Oberfläche von CORE starten. In der folgenden Abbildung ist der Ablauf veranschaulicht.



```
tobias@tobias-VirtualBox: ~/Desktop
tobias@tobias-VirtualBox:~/Desktop$ sudo service core-daemon status
● core-daemon.service - Common Open Research Emulator Service
   Loaded: loaded (/lib/systemd/system/core-daemon.service; disabled; vendor
   Active: failed (Result: exit-code) since Thu 2021-01-14 12:03:32 CET; 24s
   Process: 39647 ExecStart=/usr/local/bin/core-daemon (code=exited, status=15)
   Main PID: 39647 (code=exited, status=15)

Jan 14 12:03:07 tobias-VirtualBox systemd[1]: Started Common Open Research Emul
Jan 14 12:03:07 tobias-VirtualBox core-daemon[39647]: 2021-01-14 12:03:07,889 -
Jan 14 12:03:07 tobias-VirtualBox core-daemon[39647]: 2021-01-14 12:03:07,901 -
Jan 14 12:03:07 tobias-VirtualBox core-daemon[39647]: 2021-01-14 12:03:07,917 -
Jan 14 12:03:32 tobias-VirtualBox core-daemon[39647]: 2021-01-14 12:03:32,654 -
Jan 14 12:03:32 tobias-VirtualBox core-daemon[39647]: 2021-01-14 12:03:32,654 -
Jan 14 12:03:32 tobias-VirtualBox systemd[1]: Stopping Common Open Research Emu
Jan 14 12:03:32 tobias-VirtualBox systemd[1]: core-daemon.service: Main process
Jan 14 12:03:32 tobias-VirtualBox systemd[1]: core-daemon.service: Failed with
Jan 14 12:03:32 tobias-VirtualBox systemd[1]: Stopped Common Open Research Emul
tobias@tobias-VirtualBox:~/Desktop$ sudo service core-daemon start
tobias@tobias-VirtualBox:~/Desktop$ sudo service core-daemon status
● core-daemon.service - Common Open Research Emulator Service
   Loaded: loaded (/lib/systemd/system/core-daemon.service; disabled; vendor
   Active: active (running) since Thu 2021-01-14 12:04:07 CET; 1s ago
   Main PID: 39718 (core-daemon)
   Tasks: 7
   Memory: 33.0M
   CGroup: /system.slice/core-daemon.service
           └─39718 /home/tobias/.cache/pypoetry/virtualenvs/core-NMEWF6HO-py3
Jan 14 12:04:07 tobias-VirtualBox systemd[1]: Started Common Open Research Emul
Jan 14 12:04:07 tobias-VirtualBox core-daemon[39718]: 2021-01-14 12:04:07,569 -
Jan 14 12:04:07 tobias-VirtualBox core-daemon[39718]: 2021-01-14 12:04:07,581 -
Jan 14 12:04:07 tobias-VirtualBox core-daemon[39718]: 2021-01-14 12:04:07,582 -
tobias@tobias-VirtualBox:~/Desktop$ core-gui
Connecting to "core-daemon" (127.0.0.1:4038)...connected.
```

Abbildung 4: Start der CORE-Oberfläche

Das User Interface von CORE zu erklären würde im Rahmen dieses Dokuments zu weit führen, allerdings gibt es eine ausführliche englische Dokumentation zu den einzelnen Knotentypen, dem User Interface und vielen weiteren Themen direkt von CORE. Diese ist unter dem folgenden Link zu finden: <https://coreemu.github.io/core/>

4.1.2 Aufbau eines Netzwerks mit CORE

Die Netzwerke in unseren Szenarien bestehen der Einfachheit halber ausschliesslich aus Knoten vom Typ Router, welche durch physische Links verbunden sind. Das Verwenden von anderen Knotentypen und Wireless LAN ist natürlich möglich, aber nicht von uns getestet.

Wireless LAN in CORE unterstützt sogenanntes Mobility Scripting, wodurch sich Knoten zur Laufzeit aus der Verbindungsreichweite entfernen können, um Verbindungsabbrüche zu erzwingen. Diese Verbindungsabbrüche in CORE haben **keine** Auswirkung auf die Verbindung der ION Nodes.

4.1.3 Integration von ION

Die Integration von ION in CORE erfolgt in drei einfachen Schritten, die auf jedem Knoten ausgeführt werden müssen.

- Einrichten eines UserDefined-Services zum Ausführen eines Setup-Skripts
- Zuweisen der von ION benötigten Verzeichnisse
- Anlegen des Setup-Skripts

Zunächst muss ein Service eingerichtet werden, der ein Setup-Skript ausführt, welches dann ION auf dem Knoten startet. Diesen legt man wie folgt an: Doppelklick auf den Knoten → Services → UserDefined. Im Tab 'Files' den File name `setup.sh` (ohne `)` eingeben und auf den Button neben der Eingabeleiste klicken. 'Use text below for file contents' auswählen und in die Box darunter das folgende Skript kopieren.

```
dirn='dirname $SESSION_FILENAME'
sh $dirn/nodeSetup.sh
```

Die Backticks(`) um `'dirname [...]` können evtl. nicht richtig aus diesem Dokument kopiert werden und müssen manuell gesetzt werden.

Als nächstes müssen die Verzeichnisse zugewiesen werden. Dazu auf den Tab 'Directories' wechseln, unten rechts auf den 'Verzeichnis hinzufügen' Button klicken und `/var/ion` hinzufügen.

Zuletzt muss dem Knoten noch mitgeteilt werden, dass das in Schritt 1 angelegte `'setup.sh'` Skript ausgeführt werden soll. Dies geschieht im 'Startup/Shutdown' Tab. In die Zeile unter 'Startup commands' `'sh setup.sh'` schreiben und auf den 'Hinzufügen' Button klicken. Danach kann man die Einstellungen durch Klick auf 'Apply' anwenden und der Knoten ist konfiguriert.

4.1.4 nodeSetup.sh

Das Skript `'nodeSetup.sh'` wird von CORE ausgeführt sobald ein Knoten gestartet wurde und sorgt dafür, dass auf den Knoten ION läuft. Es muss immer im gleichen Verzeichnis liegen wie die `¡Szenarioj.imn` Datei. Für die genaue Verzeichnisstruktur, kann man jedes beliebige Szenario hernehmen, die Struktur ist immer gleich.

Wir empfehlen ein schon vorhandenes `nodeSetup.sh` Skript aus einem Szenario-Ordner zu kopieren und zu modifizieren. Der Teil in dem verschiedene ION-Services gestartet werden, ist die if-else-Abfrage nach `ionstart -I n$IPN_NODE_NUMBER.rc >> $LOG`. Sollte zudem eine Visualisierung der Anzahl von Bundles auf jedem Knoten gewünscht sein, muss sichergestellt werden, dass unter der if-else-Abfrage die Zeile `sh bundlecount.sh $IPN_NODE_NUMBER` vorhanden ist. Mehr dazu im Abschnitt 'Visualisierung des Szenarios'.

4.2 Konfiguration von ION

Die Konfigurationsdateien ION nutzt um die Nodes zu initialisieren sind komplex genug um dazu eine eigene Dokumentation zu schreiben. Eine grobe Übersicht liefert der ION Deployment Guide im `ion-open-source-4.0.0` Ordner, der sich im `dtndos` Verzeichnis befindet. Ausserdem gibt es eine Serie an Videos auf YouTube, welche sich mit ION befasst. Das die Konfiguration behandelnde Video kann man unter folgendem Link finden: <https://www.youtube.com/watch?v=gEMoxbUz-jo&t=1s>. In diesem Video wird nicht erklärt, dass man die einzelnen Konfigurationsdateien auch in einer Datei zusammenfassen kann. Deshalb lohnt es sich auch, zusätzlich die Konfigurationsdateien in unseren Szenarien anzusehen.

4.3 Visualisierung des Szenarios

Jedes Szenario hat unterschiedliche Anforderungen an die Visualisierung. Für unsere Szenarien haben wir 3 Arten an Visualisierungen entwickelt.

4.4 bundlecount.sh und bundlewatch.sh

Die beiden Skripte `bundlecount.sh` und `bundlewatch.sh` zeigen die Anzahl an Bundles auf jeder Node. Dabei wird `bundlecount.sh` auf allen Nodes im Hintergrund ausgeführt und schreibt pro Sekunde einmal die Anzahl der auf der Node gespeicherten Bundles in eine Logdatei, die dann von `bundlewatch.sh` ausgelesen und in einem Terminalfenster visuell aufbereitet wird.

Damit diese Visualisierung funktioniert, müssen die beiden Skripte im Verzeichnis des Szenarios liegen. Desweiteren muss im `nodeSetup.sh` Skript auf jeder Node das `bundlecount.sh` Skript gestartet werden.

4.5 pingvis.sh

Dieses Skript wird vom `start_dtndos.sh` Skript ausgeführt, um zu verfolgen, wann die Verbindung zwischen zwei Nodes abbricht. Dazu wird die Ausgabe des Bundle Pings auf einer Node in ein Logfile umgeleitet, welches vom `pingvis` Skript dann in einem Terminal ausgegeben wird.

Auf den Nodes sind standardmässig die Watch Characters von ION aktiviert. Dies zeigt sich in der Ausgabe des Skriptes. Jedes Mal, wenn ein Bundle

losgeschickt wird, wird einmal das Zeichen 'a' ausgegeben. Bestätigt die angepingte Node den Eingang des Signals, so wird ausserdem die Response mit Round-Trip-Time ausgegeben. Gibt es einen Verbindungsabbruch, erkennt man dies an dem Fehlen ebendieser Antwort auf die gesendeten Bundles. Im Terminal tauchen dann nur noch die 'a's für gesendete Anfragen auf.

4.6 othervis.sh

Das othervis.sh Skript ist für alle weiteren Visualisierungen gedacht. Es wird zum Beispiel im Slowloris-Szenario genutzt, um anzuzeigen wann genau der Angriff startet.

Literatur

- [1] Jeff Ahrenholz. *CORE Documentation*. URL: <https://coreemu.github.io/core/> (besucht am 17.01.2021).
- [2] Gokberk Yaltirakli. „Slowloris“. In: *github.com* (2015). URL: <https://github.com/gkbrk/slowloris>.