OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
INF FAKULTÄT FÜR INFORMATIK

Industrie-Ransom 2.0:
Evaluation der Anwendung von YARA zur Erkennung netzwerkbasierter Ransomware

TAITS: DR2
13.12.2023

Bernhard Birnbaum

Inhalt

1. Thema
2. Konzept
   a. Werkzeugauswahl/Referenzen
   b. Methodik
3. Umsetzung
   a. Wrapper-Script
   b. YARA-Regeln
   c. Module
4. bisherige Ergebnisse
5. Aussicht

1. Thema

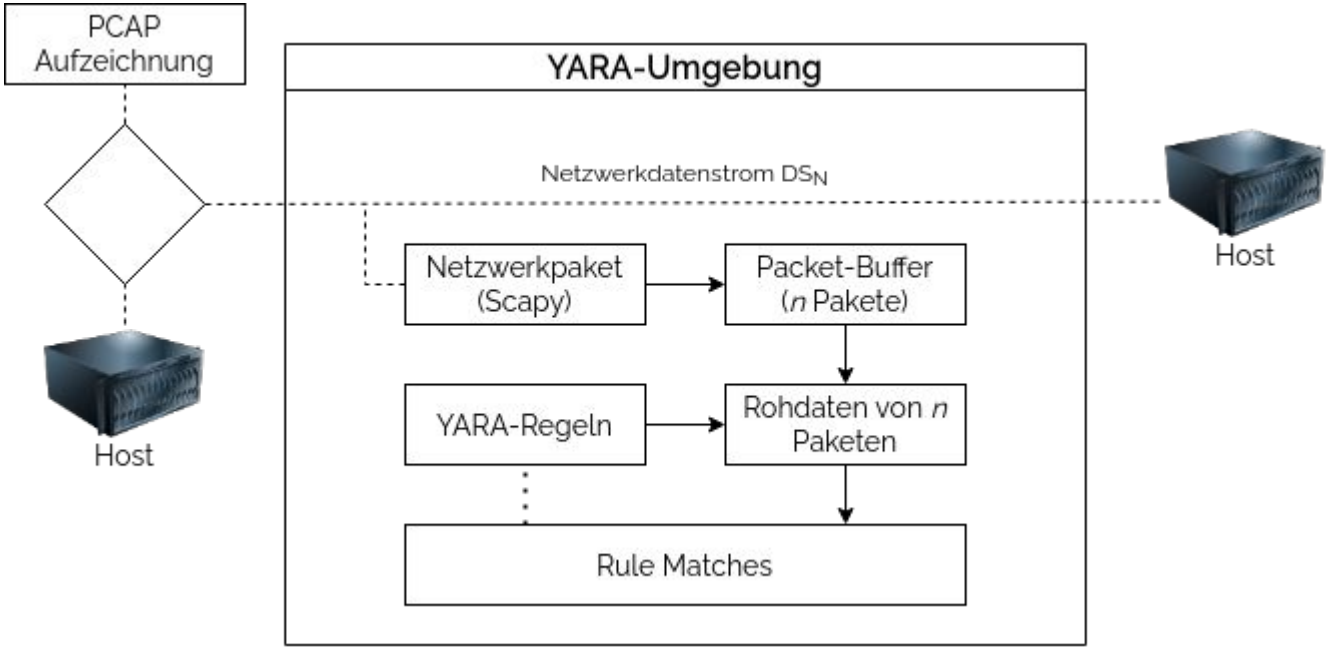
- Untersuchung von netzwerkbasierter Ransomware im industriellen Umfeld
- Systematisierung mit YARA
  - Beschreibungssprache, um Malware zu identifizieren und zu klassifizieren
  - nutzt Regeln (ähnlich wie Pattern-Matching), um schädliche Inhalte zu erkennen

→ **zentrale Fragestellung**:
  - Inwiefern kann YARA zur Detektion von (Industrie-)Ransomware im Netzwerkdatenstrom verwendet werden?

2. Konzept: Werkzeugauswahl/Referenzen

Werkzeug/Referenz	Beschreibung	Link
Wireshark	Netzwerkanalyse-Tool (v. 4.2.0)	https://www.wireshark.org/
Python	Python. (v. 3.11)	https://www.python.org/
Scapy	Netzwerk-Capabilities mit Python-Integration (v. 2.5)	https://scapy.net/ https://scapy.readthedocs.io/en/latest/
YARA	YARA-Regelsystem mit Python-Integration (v. 4.4.0)	https://github.com/VirusTotal/yara https://yara.readthedocs.io/en/stable/
ICS Datasets mit diversen Papern	Aufzeichnungen von Netzwerkverkehr	https://gitti.cs.uni-magdeburg.de/klamshoeft/ics-datasets

2. Konzept: Methodik



3. Umsetzung: Wrapper-Script

- Docker-Umgebung: Ubuntu 23.10 mit Python, Scapy, YARA
- Parametrisierung: *Packet-Buffer-Size*, *Packet-Filter* und *Pfad zu YARA-Regeln*
- Unterstützt Lesen von aufgezeichneten PCAPs, aber auch Echtzeit-Paketerfassung
- Packet-Buffer mit Deque realisiert; neue Pakete vorne, alte hinten
- Netzwerkpakete im Buffer werden sequentiell in Rohdaten umgewandelt
- YARA-Regeln werden auf Rohdaten angewandt

3. Umsetzung: YARA-Regel (Query-Flooding)

```
rule modbus_query_flooding {
        strings:
        $modbus_query_request = { 00 80 F4 09 51 3B 00 0C 29 E6 14 0D 08 00 45 00 00 34 [2] \
                                  40 00 40 06 [2] AC 1B E0 32 AC 1B E0 FA [2] 01 F6 [4] [4] 50 \
                                  18 72 10 [2] 00 00 [2] 00 00 00 06 01 06 00 06 00 00 }
        condition:
        #modbus_query_request >= 3
}
```

→ *Parametrisierung*: -pbs 3

3. Umsetzung: YARA-Regel (Kochvorgang I)

```
import "console"
import "floating"

rule opcua_kochvorgang_temperature_exceeds_50 {
        strings:
        $opcua_writerequest = { 4D 53 47 46 58 00 00 00 [4] 01 00 00 00 [8] 01 00 A1 02 02 00 \
                                00 [4] [8] [4] 00 [3] FF FF FF FF A0 0F 00 00 00 00 00 01 00 00 \
                                00 01 04 29 00 0D 00 00 00 FF FF FF FF 03 0A [4] 00 00 00 00 }
        condition:
        #opcua_writerequest > 0 and floating.float32(@opcua_writerequest[1] + 80) >= 50
}
```

3. Umsetzung: YARA-Regel (Kochvorgang II)

```
rule opcua_kochvorgang_temperature_difference_exceeds_5 {
        strings:
        $opcua_writerequest = { 4D 53 47 46 58 00 00 00 [4] 01 00 00 00 [8] 01 00 A1 02 02 00 \
                                00 [4] [8] [4] 00 [3] FF FF FF FF A0 0F 00 00 00 00 00 01 00 00 \
                                00 01 04 29 00 0D 00 00 00 FF FF FF FF 03 0A [4] 00 00 00 00 }
        condition:
        #opcua_writerequest >= 2 and (floating.float32(@opcua_writerequest[1] + 80) - \
                                      floating.float32(@opcua_writerequest[2] + 80)) > 5
}
```

→ *Parametrisierung*: -pbs 50

3. Umsetzung: Module

- Möglichkeit, eigene Datenstrukturen und Funktionen zu definieren, welche in YARA-Regeln verwendet werden können
- in C geschrieben, werden bei Kompilierung von YARA eingebunden
- prinzipiell möglich, beliebig komplexe Regeln zu schreiben (turing-vollständig)
  → Frage besteht also eher darin, wie viel Zeit und Aufwand investiert werden sollte.
- es gibt Standard-Module, z.B. “console”, “hash”, “math”, …
- “math” bietet z.B. Funktionen wie
  - *entropy*(*offset*, *size*)
  - *mean*(*offset*, *size*)
  - *deviation*(*offset*, *size*, *mean*)
  - *min*(*int*, *int*)
  - *max*(*int*, *int*)
  - …

  → evtl. auch interessant für Stego-Detektion..?

3. Umsetzung: Modul (Floating)

- für Regeln, welche auf Datenwerten an bestimmten Positionen basieren (z.B. Kochvorgang), bietet YARA standardmäßig **nur für Ganzzahlen** entsprechende Funktionen an:
  - int8(offset), int16(offset), int32(offset), uint8(offset), …
- Auszug aus *Floating.c*:

```
define_function(float32) {
        int64_t offset = integer_argument(1);
        if (block_size > offset + 4) {
                float* reinterpreted_numeric = (float*)(block_data + offset);
                return_float(*reinterpreted_numeric);
        } else {
                printf("\n[float32] WARNING: Given offset exceeds block size, can not convert!\n  \
                  Returned -1 may result in broken rules!\n");
        return_float(-1);
        }
}
```

4. bisherige Ergebnisse

- Was YARA detektieren kann:
  - einzelne Pakete mit schädlichen Werten
  - abnormale Werteänderungen über mehrere Pakete hinweg
  - Spam/Flooding von Paketen
- Was YARA nicht (ohne Weiteres) detektieren kann:
  - Zeitintervalle zwischen Paketen können nicht validiert werden (bsp. für Abweichungen im Modbus-Polling-Intervall interessant)
    → YARA-Regeln werden immer sofort ausgewertet; evtl. Workaround m.H. von Zeitstempeln zwischen Paketen im Packet-Buffer möglich
  - verschlüsselte Daten können mit Pattern-Matching nicht bearbeitet werden
    → YARA im Warden-Szenario könnte dies allerdings

5. Aussicht

- Kann YARA
  - Manipulation durch MITM-Angriff erkennen?
  - Metasploit Meterpreter-Kanäle zum Verschieben von Daten im Netzwerk erkennen?
  - LSB-Stego in Modbus-Registern erkennen?
  - …
- Einschätzung zu YARA-Modulen: Bis zu welchem Punkt ist es sinnvoll und praktikabel, eigene C-Module für entsprechende Problematiken zu schreiben?

Danke für Ihre Aufmerksamkeit!