

Expect the Unexpected

Probleme bei der Analyse mit typischen Perl-Modulen

Steffen Ullrich, genua
<http://noxxi.de/talks.html>



- Perl wird in gefährlichem Umfeld eingesetzt
 - Crawlen von untrusted Websites
 - Analyse von Mails auf Viren, Spam...
 - Handling von Uploads (Typ, Malware...)
 - Eingabeverification bei Web-Applikationen (kein Thema hier)



- .. HTTP: DOS durch Kompressionsbomben
- .. TLS: Fehlende Validierung von Zertifikaten
- .. Mail/MIME: Bypass durch Mehrdeutigkeiten
- .. Uploads, Attachments: Erkennung Filetyp



- Perl hat Vielfalt von HTTP-Bibliotheken
 - libwww (LWP::UserAgent)
WWW::Mechanize baut darauf auf
 - Mojolicious (Mojo::UserAgent)
 - HTTP::Tiny, Furl, AnyEvent::HTTP,
- Einsatz i.A. ohne spezielle Sicherheitsvorkehrungen wie ulimit



```
$ua = LWP::UserAgent->new;  
$res = $ua->get('http://...');  
print $res->content;
```

```
GET / HTTP/1.1  
TE: deflate, gzip; q=0.3  
Connection: TE, close ↑  
...
```

TE macht sonst keiner



```
GET / HTTP/1.1  
TE: deflate, gzip; q=0.3  
Connection: TE, close  
...
```

```
HTTP/1.1 200 ok  
Transfer-Encoding: deflate  
Transfer-Encoding: deflate  
Transfer-Encoding: deflate  
Transfer-Encoding: chunked  
...
```

beliebige Tiefe

Dekompression am Stück und im Hauptspeicher

743 Byte → 4 GByte



```
$ua = WWW::Mechanize->new;  
$res = $ua->get('http://...');  
print $res->decoded_content;
```

```
GET / HTTP/1.1  
TE: deflate, gzip; q=0.3  
Connection: TE, close  
Accept-Encoding: gzip  
...
```

← gzip an per Default

```
HTTP/1.1 200 ok  
Content-Encoding: gzip  
Content-Encoding: gzip  
Content-Encoding: gzip  
...
```

← beliebige Tiefe
Dekompression am Stück
und im Hauptspeicher
785 Byte → 4 GByte



- nur ein Level der Dekompression
 - Chunkweise Dekompression ohne Bombenschutz
 - maximale Readsize 131072
x 1000 → 140 MB



- Furl:
 - ein Level Dekompression
 - Readsize 10kB (→10MB), aber sammeln der Ausgaben
- HTTP::Tiny, AnyEvent::HTTP
keine Dekompression
- Python Requests (urllib3)
 - ein Level Dekompression
 - Readsize 1MB (→ 1GB)



- Bots werden vielfach genutzt um auf potentielle Malware zuzugreifen
 - WebCrawler, Follup-Up von Virustotal
 - Online-Validierung bei BlueCoat, Trend-Micro...
- Bot versuchen oft sich durch „korrekten“ User-Agent zu tarnen
 - z.B. Bluecoat kopiert UA vom originalen Request
 - Trotzdem leicht zu entdecken, z.B. durch Werte in TE, Accept-Encoding, Accept, Source-IP ...



- Keines der Module rechnet mit Kompressionsbomben.
- Meistgenutztes LWP am leichtesten angreifbar.
TE sollte entsorgt werden, zumal kaputt.
Stacking von Content-Encoding entfernen.
- Bots sind leicht zu entdecken.
Füttern mit Falschinformationen möglich.
Ändern des User-Agent nicht ausreichend.



- Validierung per Default aus:
Mojo::UserAgent, Mail::Sender, Net::FTPSSL,
Email::Send::SMTP::Gmail
LWP abhängig von Environmentvariablen
- Validierung permanent aus:
File::HTTP, Net::SMTP::TLS_ButMaintained
- Beschränkung TLS Version:
SpamAssisin spamd (SSL 3.0, max TLS 1.0)
IMAP::Client, Mail::Sender (TLS 1.0 default)
kaputter Default in Net::IMAP::Simple, Net::SMTP::TLS



- Diverse Module unterdrücken die standardmäßige Validierung der Zertifikate
Funktioniert ja trotzdem, sogar besser als erwünscht.
- Einige Module erzwingen TLS Versionen, die nicht mehr zeitgemäß sind



- Mail ist Text
MIME bildet Struktur auf Text ab
- Analyse von Mails u.a.:
 - Textanalyse in SpamAssisin
 - Extraktion Attachments zum Virenschannen in Amavisd, MimeDefang, Anti-Spam-SMTP-Daemon
 - Interface zu Virenschannern in qpsmtpd



- Parsen der MIME-Struktur
 - MIME-Tools (MimeDefang, Amavisd, SpamAssisin)
 - Email::MIME (Anti-Spam-SMTP-Proxy)
 - ad-hoc (qpsmtpd)
- Content-Transfer-Encoding
 - MIME::Base64, MIME::QuotedPrint
 - MIME-Tools: MIME::Decoder::*
basieren auf MIME::Base64 und MIME::QuotedPrint
 - eigene base64-Variante mit `unpack(„u“, ...)` in SpamAssisin



```
Content-type: multipart/mixed;  
  boundary=foo
```

```
--foo
```

```
Content-type: text/plain
```

```
Das ist der erste Teil
```

```
--foo
```

```
Content-type: application/octet-stream
```

```
Content-Transfer-Encoding: base64
```

```
Content-Disposition: attachment;
```

```
  filename=file.txt
```

```
RGFzIGlzdCBkZXIgendlaXRlIFRlaWwK
```

```
--foo--
```



- Fehlerhafte Interpretation des MIME führt zu Umgehung der Analyse
- Mehrdeutigkeiten möglich durch
 - mehrere Multipart Boundaries
 - mehrere Content-Transfer-Encoding Header
 - leicht ungültiges Base64 Encoding
 - leicht ungültiges Quoted-Printable Encoding
 - ...



Mehrdeutige Boundary

Content-type: multipart/mixed; boundary=foo

Content-type: multipart/mixed; boundary=bar

foo

MIME-Tools
Email::MIME
Thunderbird
AOL, gmail

bar

mutt
Yahoo, GMX*, live.com

Content-type: multipart/mixed;

boundary=foo; boundary=bar

foo

Thunderbird, mutt
Yahoo, gmail, GMX

bar

MIME-Tools
Email::MIME
AOL, live.com



```
Content-type: multipart/mixed;  
boundary*0=foo
```

Multipart

MIME-Tools
Thunderbird, mutt
gmail,AOL, live.com

Plain

Email::MIME

Leer

GMX, Yahoo

```
Content-type: multipart/mixed;  
boundary*=' 'fo%6F
```

Multipart

Thunderbird, mutt
gmail,AOL

Plain

Email::MIME
live.com

Leer

MIME-Tools
GMX, Yahoo

```
# qpsmtpd/plugins/virus/bitdefender  
unless ($content_type  
  && $content_type =~ m!\bmultipart/.*\bboundary="?([\^"]+)!i)  
{  
  $self->log(LOGERROR, "non-multipart mail - skipping");  
}
```



Content-Type: `text/plain`

Content-Type: `multipart/mixed; boundary=foo`

text

MIME-Tools
Email::MIME
Thunderbird
AOL, gmail

multipart

mutt
live.com



Content-Transfer-Encoding: **base64**

Content-Transfer-Encoding: **quoted-printable**

base64

MIME-Tools

Email::MIME

Thunderbird

Yahoo*, AOL, gmail, GMX

quoted-printable

mutt

live.com



- .. 3x8 Bit (binär) → 4x6 Bit [0-9A-Za-z_!]
- .. Auffüllen mit '=' am Ende

0x58	0x35	0x30
01011000	00110101	00110000
W	D	V
		P

0x58	0x35	
01011000	00110101	<u>00000000</u>
W	D	V
		=

\x58\x35\x30	→	WDVP
\x58\x35	→	WDV=
\x58	→	WD==



Reaktion auf invalides Base64

```
# Das ist ein Test
```

```
RGFzIGlzdCBlaW4gVGVzdA==
```

```
# ungültige Zeichen werden i.A. ignoriert
```

```
RGFzIGlzdCB %%% laW4gVGVzdA==
```

```
# '=' sollte nur am Ende stehen
```

```
RGFzIGlzdCB = laW4gVGVzdA==
```

Abbrechen [Das ist](#) MIME::Base64, Python

Ignorieren [Das ist ein Test](#) MIME::Decoder

Ersetzen [Das ist ..binär..](#) SpamAssisin

Einbeziehen [Das ist ..binär..](#) Thunderbird

Variationen des Themas erlauben alle Antivirus auf Virustotal zu umgehen, aber Thunderbird die gewünschten Daten zu liefern



```
# Über Lösung 4=5 ärgern
=DCber L=F6sung 4=3D5 =20=E4=
rgern
```

- Nicht-ASCII Zeichen als Hex =HH, = → =3D
- Lange Zeilen unterbrochen mit finalelem '='
Spaces zwischen '=' und newline müssen ignoriert werden.
- **Aufeinanderfolgende Spaces** müssen kodiert werden.
Machen praktisch alle falsch.



Invalides Quoted-Printable

```
# Ein übler Test  
Ein =FCbler Test
```

```
# '=' nur als =HH oder am Zeilenende erlaubt  
Ein ==FCbler Test
```

Skip 1

```
Ein =übler Test  
MIME::QuotedPrint  
MIME::Decoder  
ClamAV  
mutt  
GMX, AOL
```

Skip 3

```
Ein ==FCbler Test  
Thunderbird  
viele andere
```

Drop 1

```
Ein übler Test  
gmail
```

Skip 1 + Drop 2

```
Ein =Cbler Test  
live.com
```

Drop 1 + Skip 3

```
Ein =FCbler Test  
Python quopri
```



```
# White-Space zwischen '=' und <nl> muss  
# ignoriert werden  
# VIRUS  
VI=<space><nl>RUS
```

VIRUS

MIME::QuotedPrint
MIME::Decoder
Thunderbird
GMX, AOL
VirusTotal 8/30

VI=<space><nl>RUS

Python quopri
mutt
live.com, Yahoo
ClamAV, snort



- Keiner der Bibliotheken taugt für Einsatz in unfreundlichen Umgebungen
 - gehen von „üblichem“ gutartigen Input aus
 - Mehrdeutigkeiten werden still ignoriert
 - Ziel ist Robustheit, nicht Sicherheit
- Trotzdem werden viele für sicherheitskritische Aufgaben herangezogen
- Kommerzielle Produkte (AV, Firewalls) sind hier allerdings kaum besser



- Basierend auf Endung
Unterschiede je OS und installierter Software
- Basierend auf magic Bytes
 - File::LibMagic
File::MMagic
File::Mimeinfo::Magic
file(1)
 - z.B. in amavisd: Lib::Magic bzw. file(1)
- Problem: liefern alle nur einen Typ zurück



- ist das ZIP, DOCX, ODT, JAR, EPUB ... ?
- Polyglot-Dokumente
 - GZIP+ZIP: `junk.gz + malware.zip`
 - GIF+JS: `GIF89a=1;alert('p0wned');`
 - Flash+JS, ...+PDF, u.v.m
- Magic-Bytes nur Heuristik
- Interpretation evtl. abhängig vom Kontext
- daher untauglich für **sichere** Entscheidung, welche Analyse gemacht wird
Besser „normalisieren“ Inhalt, Extension...



- HTML JavaScript, CSS Kontext
`<script src=image.gif>`
- HTML festlegen der Fileendung Download
``
- MIME multiple bzw. unklare Filenamen
 - Content-Encoding: ...; name=foo.txt
Content-Disposition: ...; filename=bar.exe
 - Content-Encoding: ...;
name*0=foo; name*1=.e; name*2=xe
 - u.v.m.



- Unfreundliche Einsatzumgebungen haben spezielle Herausforderungen.
- Die meisten Bibliotheken sind nicht dafür designed.
- Anwendern sind Limitierungen nicht klar. Autoren oft auch nicht.

