# Pozzo & Lucky, The phantom Shell. Stego in TCP/IP (part-2)

by John Torakis @ securosophy.com
(*john.torakis [at] gmail [.] com*)

## Some Steganography Theory Basics

In the last post (Teaching an Old Dog (not that new) Tricks), there has been some fuzz about steganography. So before we continue to part-2 let's have a little talk about what really goes on with stego.

Stego has **2 categories**:

- We can write steganographically a Shakespeare play in an image with a number of zebras and be sure none will notice, because searching the LSB of every byte of every pixel is no sane action for anyone viewing an image. But this doesn't mean that if you look there you won't find the play. This type of stego is the "*Hidden in plain site*" stego. The whole part-1), where we pass plain data around by encapsulating it in TCP/IP headers, falls under this category.
- The second category (the one that the above Tanenbaum example really falls under) is a lot better. It uses encryption to make sure that even if you turn the image inside-out you won't see a trace of the Shakespeare play without knowing a certain secret (key?).

The other meaningful clarification is why it is superb to use **Stego** over **Encryption**, given that none really can read you in both techniques. The difference lies on that if you use **encryption**, while none can understand what you are saying (beside the authorized listener), everyone can tell that you and the listener have a **communication channel**, and also that you might be talking about something confidential (*that has to be the reason why you are using encryption*). **If you use stego none can see the communication channel.** So you aren't publicly announcing that you are communicating. A communication channel that none can imagine is a **covert channel**.

The bad news is that stego most of the time **leaves traces**. And some times very self-explanatory ones. For example, LSB stego in images creates a high number of color variations that easily can be almost a **proof of steganography usage**. Or, in my TCP/IP stego in part-1), pushing ASCII bytes **only** in random fields significantly lowers the entropy of the field data, showing a communication channel possibility, or even the communication itself, to a forensics performer. And uncovering the covert channel of a Stego just **downgrades** it to plain Encryption.

# And now for something completely different!

## Pozzo & Lucky

**Pozzo & Lucky** are 2 key characters in "Waiting for Godot". This is a Samuel Becket play, maybe the most known Samuel Becket play, and my favorite one. You can read all of it here: Act-1, Act-2 (there are just 2 acts).

**Lucky** is a servant with no beliefs, opinions or even thoughts of his own. He blindly obeys **Pozzo**, who is dragging him all over with a dog collar. He **dances** or even **thinks** whenever Pozzo commands.

In the play we have no idea **why** Lucky is so pathetic and lets Pozzo do all kind of nasty stuff to him. *There must be a covert channel between them…*

But, beside character names of an irrelevant play, **Pozzo & Lucky** is one personal project. A project that started with a bet. **"Can there exist a Remote Command Execution shell that no network device can detect and leaves no network trace?"**. I bet it can…

Well, I won my bet. This shell exists and is named **Pozzo & Lucky**…

## The Idea

The idea is almost close to the part-1) idea, except **as hardcore as it gets**. We are passing commands through **IP identification** and **TCP sequence** (ISN) fields. But, this time, we do it right…

The **Pozzo & Lucky** shell consists of 2 components. **Lucky**, which has to be installed (actually just run) on the target machine and **Pozzo** which is used to control the target machine after Lucky is installed in it.

## The Features

- Complete OS command execution (with and without output)
- Remote on-the-fly Shellcode Execution (paste and BOOM)
- File Upload/Download
- Complete immunity to .pcap file analysis, Firewall log analysis and generally analysis without OS forensics from the target machine
- Capability to simulate an nmap -sS port scan or any kind of SYN scan, or SYN flood to specific (or given) Destination Port(s)
- Works (or has to work) on Windows and Linux.
- Creates no connections. Every single packet in the same "conversation" can be send from different Source IP and to different Destination Port.

## Some Drawbacks

- Painfully slow! (Bandwidth is 5 bytes/packet, so be patient)
- As a process it has no capabilities to hide itself or get persistent. It has to be paired with a rootkit for that.
- Proxies kill it (while they don't detect it). It has to work through port-forwards though.
- Has dependencies… Scapy on Linux and Scapy with Winpcap on Windows. Both may be mitigated with a PyInstaller–py2exe–nuitka session (except maybe the damn .dll).

## Requirements

- **Needs root/admin privileges** to get installed on the target machine (due to packet crafting and sniffing needs).
- Needs Pozzo to be in the **same subnet** with Lucky (this could be the whole Internet – 2 hosts with public IPs), or at least Pozzo to have **a direct TCP port route to Lucky** (Lucky behind a Firewall with portforwarded just TCP port 21, would work if Pozzo sends packets to <Firewall_IP>:21.
- **Pozzo shouldn't be behind a NAT.** That is because the Source Port of the outgoing Pozzo packets is **meaningful to Lucky**, and NAT changes this field (as it translates it to another Source Port before forwarding with the Gateway's IP).

## The Concept

The target machine runs **Lucky**, which is basically a **packet sniffer**. It gets all packets arriving to the machine, and **decides** which of them are created by the computer running Pozzo using an algorithm described in the section "**Problem Solving**".

The crucial part is that those packets **do not establish connections in the target** (neither TCP nor "UDP"). They are TCP SYN packets that do not abuse the ~~TCP protocol in any way~~ (more on this on part-3), so they pass through protocol sanity checks (performed by security devices and packet inspectors). They also are useful packets, that **cannot be generally blocked in a network** (unlike ICMP), as this action will render the network useless (no connections will be allowed in a network that blocks SYN packets, so no SQL, web applications, FTP, etc – you get my point…).

The fishy things with those "*Pozzo packets*" is that they deliver **6 bytes of data** through **IP identification** field and TCP Sequense Number field (2 bytes + 4 bytes), in a strongly encrypted form. When Lucky encounters such a packet it extracts the 6-byte payload, splits it in a 1+5 byte form, where the first byte is an **Opcode** for the **command** to run with the next 5 bytes.

It then generates a **RST-ACK** packet, that doesn't violate the TCP protocol too, and **injects** (encrypted as well) the response of the command executed on the target, sending it **back to Pozzo**.

That SYN-RST **ping-pong** resembles a **Port Scan** a lot more than a **Remote Command Execution**, so **it doesn't get blocked by IDS/IPS**, as there are **no signatures** due to encryption (and they rarely look at layer 3-4 headers). A really well configured firewall device, with a configuration aware of each host usage (this is an SSH Server – allow just 22) may mitigate **Pozzo & Lucky**, but I haven't seen a lot of them!

## Problem Solving

Some problems have risen from part-1). Here I explain how I tackled them.

### Surpassing the entropy problem

The problem with entropy is that when we could use any of the **256 bytes** in every byte place in a random field, we just use a byte from the **printable ASCII** list, while generally excluding the Upper Case letters and numbers. This made the random fields contain very predictable data, thous **lowering the data entropy**.

The solution to this is Encryption. But we need a cipher with 6 byte blocks, or a stream cipher. And most of all, *we need to do it with style…* So I managed a custom **One Time Pad Scheme** based on plain **XOR** and **SHA512**. A simple one, that doesn't lack style at all!

### The OTP Scheme

You get a passphrase, SHA512 it and get a **key**. With this key **we XOR data**, 6 bytes of data. The XORed data is **securely encrypted** as the key **is a one-way function** of the passphrase, which is our secret. To encrypt the next 6-byte chunk, we SHA512 the **current key** and reXOR. This way we **never XOR with the same key**, which eliminates the possibility of "cryptanalysis" using the *known-plaintext* technique. We also eliminate the possibility of prediction of the next keys, as **even if we encrypt all the time the same 6 bytes** (say "ls -la"), the key portion that can be retrieved each time is 6 bytes. With 6 bytes we **lack enough information to produce the next key**, as a whole key is of 512bits (64 bytes) long.

Plus, this way, by having the possibility to XOR with any possible byte (SHA512 returns a byte sequence containing all kinds of bytes) we get encrypted bytes in the whole 256 byte-range. And with even possibility each one… This means **Entropy close to 1**. This means **data seemingly random**.

### Surpassing the Identity Problem

"*Who is your master?*". An RCE shell has to **know** how to answer this question. You can run commands remotely, that's a good thing, but you **MUST be the only one that can do that**. The shell must identify **your** packets from packets of others. And to enclose an **IP check** in the shell agent program you have to **hardcode** your **IP** or a **domain** in it. You **got caught** just by thinking of it, unless you use **techniques used in Exploit-Kits**, like rapidly changing sub-domain names, and other things *that lack style*, and get **caught and analyzed eventually**!

Last time (part-1), if you haven't read it by now, do me a favor…) **we forgot about a field** we can control in TCP and none cares in a port scan. The **Source Port**. "*OK, you will think, let the packet come from port xxxx and then this is a packet to decrypt and execute*". Well, yes, but it *lacks style too*. So here goes:

### Solving the "Who is your master?" problem

The thought of Source Port checking is correct up to a certain point. There is just a big catch. It is **implemented** as easily as it is **observed** by an analyst. If you get a .pcap file, with all kinds of **Destination Ports** and one **Source Port** (even with multiple Source IPs) you might suspect something.

- Why a port scanner need to allocate port 23456 in multiple systems?

- Is it hardcoded to do so?
- Do you know any such port scanner?
- Is it a common behavior?
- Googling port 23456 returns nothing.

So there is something fishy going on.

In **Pozzo & Lucky**, we check the Source Port of the packets, but **we don't expect it to be the same all the time**. There is a cycling algorithm for that too, just like the OTP Scheme above (it actually uses it).

A Source Port field **contains 2 bytes**. So **4 hex digits**. We initialize the first (Most Significant) digit depending on a **given passphrase** (it has to be more than 8 – to always get high ports). Then we SHA512 the passphrase and get the first **3 hex digits** of the hash. **Concatenating them with the initial hex digit** gives as **4 hex digits**, or **2 bytes**. Then we **cycle the hash**, by rehashing it and **generate the next port**.

This technique gives us **different port numbers**, in a **totally unpredictable sequence** for someone that doesn't have the passphrase. Only the **agent-program (Lucky)** and the **client (Pozzo)** know the next correct port to communicate and the **possibility of a stray packet** with the correct Source Port is 1/65536, **so quite slim**.

## Surpassing Inconsistent States (or the Dog Collar)

**While slim**, the possibility of the agent-program to receive a Correct Source Port stray packet (not created by the client-shell) **is existent**. If this happens, the agent is going to cycle to the next source port, cycle the encryption key, try to decrypt a packet that contains no stego and get gibberish that is gonna try to execute. **A total out-of-control mess**.

And it is **out-of-control** as the client **knows nothing about the key cycles** happened and will continue to encrypt with keys **no longer recognized by the agent** and **send from a Source Port that the agent no longer hears** from.

That means that **we lost it**. We lost RCE to the pwned machine. We have to **re-exploit** it and use another **post-exploitation tool**… But, remember, **Pozzo** was holding **Lucky** by a **Dog Collar**. He was **able to reclaim him anytime**.

## The Dog Collar Implementation

There is of course a **safety mechanism** to prevent such tragedies. In the OTP Scheme a special **Control Key** is stored that does not get cycled. There is also a **Control Source Port** that the agent always accepts packets from and **decrypts them with the Control Key**. If such a packet contains **a special RST payload** then the **OTP key** and the **Source Port cycling** mechanism both **reset**.

That means that the whole communication can start **from the beginning** if jammed, without **leaving any unencrypted trace**.

## A Long Payload is Longer than 5 bytes

There are commands like "find / -name 'flag' 2 > /dev/null" that **exceed the 5 byte limit** (+1 byte the opcode) of a single packet. Those commands should be **chunked and delivered in multiple packets**. And Lucky has to understand that the "find" (notice the space – 1 byte!) isn't the whole command and it has to wait for next packets to arrive.

There is also the case of "head -1 /etc/shadow" (to get just the hash of the root password). This command produces an output that reaches and exceeds **100 bytes**. And they have to get delivered **back** to Pozzo. All of them. And Pozzo has to know **when to wait** for more output, and when the **whole payload** is delivered. Also Lucky never sends packets that **aren't responses** to packets (remember only RST-ACKs).

## The Protocol within a Protocol

If you can use Opcodes, then you can be stateful, and that means that you can **know when to wait for more**. There are Opcodes that declare that "*more is coming, don't execute just yet*". Opcodes that declare "*this data is part* *of a command*", and Opcodes that declare "*this data is the last of command. Execute it now*". It resembles the TCP chunking algorithm just without using data offsets. Ain't no time and **bandwidth** for data offsets anyway! The OTP scheme ensures that if a packet is lost no later packet can be decrypted, so **no partially executions** are possible, and inconsistent states do get resolved.

### What about Lucky's long responses?

Lucky **never sends a packet that is not a Response**... That means that it has to inform Pozzo that he needs to talk. Then Pozzo **starts sending** random data (with a "talk" Opcode), only to accept meaningful responses. Lucky also declares when there is nothing left to say. And "*the rest is silence*" (till the next command).

### Shellcode execution kills Lucky

When shellcode is delivered, in Linux is executed with the above **ctypes** snippet:

```
libc = CDLL('libc.so.6')          # Loads libc
sc = c_char_p(shellcode)          # creates a C string with shellcode
size = len(shellcode)             # gets shellcode's length (used later)
addr = c_void_p(libc.valloc(size))  # allocates bytes of heap memory equal to the shellcode length.
memmove(addr, sc, size)           # copies shellcode from stack variable(pointer) sc to heap memory that was just allocated
libc.mprotect(addr, size, 0x7)    # disables NX protection of data memory
run = cast(addr, CFUNCTYPE(c_void_p)) # casts the pointer to shellcode in heap to a function pointer
run()                             # jumps shellcode function pointer - runs the shellcode
```

Which copies it into heap memory, unlocks the NX protection for this memory chunk and jumps to it. So Lucky stops executing as EIP now points to the shellcode. No return is possible. Lucky will terminate whenever the shellcode terminates...

### Just Fork It!

```
p = Process(target=run)   # run the shellcode as independent process
p.start()
```

instead of plain:

```
run()
```

Took me a good to half hour of screen-staring...

In Windows the *CreateThread()* works as intended. That was a blessing as EIP can't be tracked in Windows. None is really sure were EIP is in any given time. Not even its developers.

## It's Show Time!

## The Test

### Start Lucky

```
# ./lucky.py mypassphrase
```

And Lucky starts happily. Uses the passphrase to create the OTPs and waits patiently...

### Connect Pozzo

```
# ./pozzo.py target_ip mypassphrase
```

## A real Infection

```
cp lucky.py /usr/sbin/X
printf "@reboot /usr/sbin/X --rootless -noreset\n" > /etc/crontab
```

Remember, the original X executable is **located at /usr/bin** directory… I personally don't believe that a Sys Admin **would realize** that this process is a phony in a plain "ps aux". Maybe an **optimistic 4/10** of Sys Admins would catch this. You need tools to catch this guy, if you aren't an observant geek!

And the **passphrase** for this Lucky instance is (yes, you guessed it!) "*–rootless*" (argv[1]). You can come up with **any switch-like passphrase** and use it. I know **no man alive** that knows **all the X switches**… And there will never be a man that will read X's man (page)!

(Here we hacked a mind, not a PC. In my humble opinion that's what "Hacking" is all about)

Passphrases can also be **hardcoded** in **lucky.py**, but this *lacks style even more*! And apart from the style part, *strings* command will return nothing (in a PyInstaller'd Lucky) if the passphrase is passed as an argument. **Hidden in plain site**.

## Video Mode ON

### The OS Shell

Here I run some linux commands in the **Pozzo & Lucky** while sniffing with tcpdump.

Video Link

### The Shellcode (ASM) Shell

Here I **remotely** run some **shellcode** I found **online**. The connection **broke the first time** I tried to deliver the shellcode so I **restarted Pozzo** to force a **Reset Packet** and get everything working again.
Video Link
I also demonstrate that Lucky **does not die** after the shellcode termination by using the OS shell again.

## Video Mode OFF

## Concluding…

This project is **closed-source** at the moment as it is a part of a **personal research** which **isn't finished** yet. Generally the whole idea has started to have an **academical perspective** as there are papers like "Embedding Covert Channels into TCP/IP" (*Murdoch & Lewis*, 2005) – *I told you the idea isn't new*, that have to be **cross checked** (those guys propose algorithms that **bust IP/TCP stego**).

Additionally anyone can treat this article as a **proposal** for a tool and start writing his/her own **implementation**. My techniques aren't the best (*while full of style*), and I am sure that some things can be done better. I learned a lot of things while writing **Pozzo & Lucky**, don't lose the opportunity to do the same. And there are things (maybe a lot of things!) to be done! Here are some:

- Write such a tool in an **ASM compilable language** (C++ maybe…)! It will be an **overkill** tool. As there will

be **no dependencies** (and if there are you can always use `--static`).
- Use another (innocent looking) **protocol**. What about **ARP**. ARPs aren't blocked unless the Network admin is a madman and has locked ALL switch ports to MACs. And even if this happens, a **Gratuitous ARP** could be received by everyone in a LAN. I see some potential here…
- Go for implementations for the **pseudo-code given in the above paper**. There can be Covert Channel **filters**. There can be a **classification model** to provide **possibilities** about whether a packet **contains** Stego. I mean, why aren't there **such things around?**
- I would really like to see a **PF-Sense plugin** for **Stego filtering**.
- The list goes on (without me)…

# Part 3?

## Sure, thanks for asking!

It will contain my research on **detection** and **mitigation** of such **techniques**. Going for an article targeted to Blue Teams!

There are some handles right now that might get us caught!

The entropy of the **TCP Sequence Field** is as high as **/dev/urandom**'s entropy for the same number of bytes, sure, but what about **distributions**? The ISNs are created (by Operating Systems) using **time** as a "seed", they **aren't entirely random**. That means that they **inevitably have a distribution**. Does Pozzo & Lucky create ISN's that resemble the same **distribution**? Most likely **NO**.

- Can we **determine** if a **packet stream** contains Stego using **this info**?
- If Yes, we need **many packets** (many values to identify the distribution).
- H**ow many**?
- How much **data has to leak** before we catch the culprit?

Research Everyone! Next time we aren't gonna fire up "**Scapy**" but "**Scipy**"!
Next time there will be **Fuction Curves** and **Integrals**, along with **Firewall and IDS logs**! I can't thing of anything better (girlfriends are pretty neat too)!

## Keep tuned…

(Holly Cows, everything we can think of exists ! Fitter, for example! That's why Python is my Business – and Business is good)

# To Be Continued…