# A peek under the Blue Coat

## ProxySG internals

Raphaël Rigo / AGI / TX5IT

Black Hat Europe - 2015-11-12

AIRBUS GROUP

# Outline

# Outline

# What? Why?

## Blue Coat ProxySG?

- enterprise (Web) proxy
- one of the most deployed in big companies
- lots of complex features:
  - URL categorization (WebSense and others)
  - video streaming / instant messaging specific handling
  - MAPI and SMB proxy / cache / prefetcher
  - etc.
- runs proprietary SGOS


## Why research ProxySG?

- widely used in Airbus Group
- interesting target for malicious actors: log bypass, Internet exposed, MITM, etc.
- no known previous research: unknown security level
  - security bulletins: mostly OpenSSL and Web administration interface bugs

# Research

Study objectives:

- assess the global security level
- write recommendations for secure deployment
- be prepared for forensics in case of a compromised ProxySG

Why publish?

- first public info but surely not first research
- foster research $\Longrightarrow$ better security

Today's presentation:

- raw technical results, as a starting point for research
- goes from low level (FS) to high level, following our approach
- applies to all ProxySG models and 6.x versions *up to Q1 2015*

## Getting started

Running ProxySG:

- hardware: commodity x86 CPUs, HDD, etc.
- VMware appliances

Common versions:

- 5.5: older version, EOL Aug 2014
- 6.2: previous *long term release*, EOL Oct 2015
- 6.5: latest *long term release*, recommended by BC

To get a first look, we need to access the filesystem:

- 6.? ($\geq$ 6.4): small FAT32 partition containing proprietary BCFS image
- older versions: fully proprietary disk partitionning/data (no FAT32)

# Outline

# On disk data: intro

## Hardware

Basic architecture: 3 disks (or more)

- small CompactFlash or SSD for OS (FAT32)
- 2 or more drives for data (proprietary FS)

## Filesystems

- static, *read-only* FS for OS (**BCFS**):
  - OS files
  - low level (static) configuration: kernel options, resource limits
- *cache engine* FS based on hash tables (**CEFS**) (Patent US7539818)
- *registry* in CEFS for settings

## Remarks

- unknowns:
  - CEFS structures
  - log storage format
- on-disk partition structures are very complex
- today: only static FS (BCFS) for OS files

# System disk organization (BIOS mode)

## Files on FAT32 partition

```
/sgos/boot/systems/system1
/sgos/boot/cmpnts/starter.si
/sgos/boot/cmpnts/boot.exe
/sgos/boot/meta.txt
/sgos/fbr.con
```

Both *starter.si* and *system1* use BCFS

## bootloader: *starter.si*

- 6 MiB
- basic SGOS (UP kernel, drivers, no application)
- looks up available systems
- displays GRUB-like boot menu

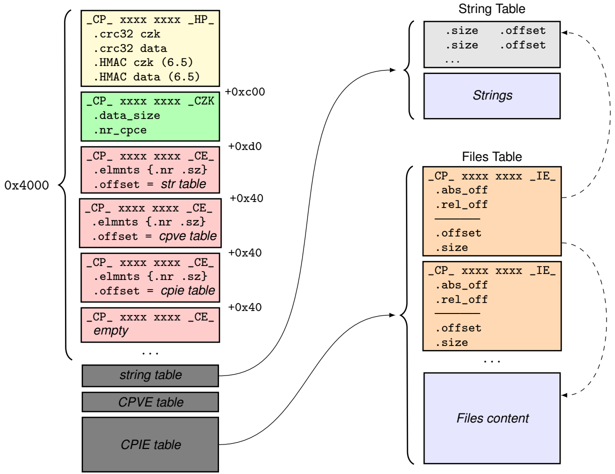
## Real OS: *system1*

- 210 MiB
- full blown OS:
  - SMP kernel
  - Web UI
  - actual applications
  - etc.

# Boot sequence (BIOS)

1. BIOS
2. MBR
3. boot sector of active partition
4. *boot.exe*, found by hardcoded sector number
5. *kernel.exe*, first file entry in *starter.si* FS
6. kernel starts *sequencer.exe*, second entry in *starter.si*
7. *sequencer.exe* parses the *main.cfg* script and starts the necessary drivers
8. *main.cfg* finally launches *starter.exe* which displays the boot menu
9. *starter.exe* loads the selected system

# BCFS (read-only FS) format




## How to extract?

1. read CPCE entries, note offsets for strings table and files table
2. parse files table (CPIE) linearly
3. get file name from strings table

## How to modify?

1. cannot increase file size
2. fix CRC and HMAC

# System image configuration variables (CPVE)

- offset and size specified by 3rd `_CP_ _CE_` entry
- modifying the variable implies fixing CRC/HMAC and reboot
- variable names can be found in *sequencer.exe*

## Structure

```
struct cpve_entry {
    uint32_t magic1; /* _CP_ */
    uint64_t unk;
    uint32_t magic2; /* _VE_ */
    uint16_t number;
    uint16_t section;
    uint32_t unk2;
    uint64_t value; }
```

## Known variables (*section*, *number*: *description*)

Section 4, `kernel`:

- 4,0: `flags`:
  - 0x8: GDB monitor enabled
  - 0x200: int3 at OS startup
  - 0x400: kernel debug logs enabled
- 4,1: `arch_flags`
  - 1: activate *Write Protect* in `cr0`
- 4,3: `console_speed` (in bauds)

# Cache Engine FS (CEFS): writable storage

- hash-table object storage with disk backend
- mostly used for cache data:
  - web content
  - CIFS files
  - MAPI mails
  - etc.
- *regular* files are also supported, with prefix `/legacy/cache_engine/`

**Some files** (paths straight from the code, no typo)

```
.../persistent/replicated/authorized_keys
.../persistent/replicated/volatile//config/v9/registry/registry.xml
.../transient//snmp.log
.../persistent/replicated/licensing_certificate
```

# Registry: settings storage

- tree structure used for all settings
- entries are referenced by strings like “config:Authenticator:local_users”
- on-disk storage: xml file on writable CEFS

**URLs** (admin rights needed)

```
/registry/show
/registry/registry.html
/registry/registry.xml
/registry/debug
```

**Interesting CLI extensions** (cf slide 24)

```
reg-set
reg-delete
reg-list
reg-trace
```

# Outline

# OS Filesystem organization

- /
  - *.cfg
  - var/[...]/lib/lib(gcc_s|stdc++)_sgos.so
  - home/jenkins/workspace/SGOS6_sg_6_5_xx7/scorpius/sg_6_5_xx7/
    - bootchain/x86/release/
    - bin/x86_64/sgos_native/release/gcc_v4.4.2/
      - *data files*
      - stripped/
        - *libs and programs*
        - mp_cr/kernel.exe
        - storage/*drivers*.exe

# ELF files: kernel, libs, programs

Everything interesting is located in `.../stripped/`:

- `.exe`, `.exe.so` and `.so` extensions (version 5 was using PE files)
- 32 or 64 bits ELF files, depending on model (RAM size?)
- *everything* in C++, compiled with g++ with custom *sgos* target
- lots of unit tests
- more than 2600 source files referenced
- everything is stripped, but lots of external symbols
- heavy template use: AMI::Config_Data::Config_Data(AMI::Storage_Class, AMI::String_Ref const&, AMI::Shared_Ptr<AMI::Installed_Systems const> const&, AMI::Shared_Ptr<AMI::Config_General const> const&, AMI::Shared_Ptr<AMI::Shell const> const&, AMI::Shared_Ptr<AMI::SSL const> const&, AMI::Shared_Ptr<AMI::SMTP_Data const> const&, AMI::Shared_Ptr<AMI::BC_Threat_Protection const> const&, AMI::Shared_Ptr<AMI::Banner_Settings const> const&, AMI::Shared_Ptr<AMI::Policy_Settings const> const&, AMI::Shared_Ptr<AMI::Statistics_Export_Settings const> const&)

"custom" ABI in 32 bits (probably gcc called with `-mregparm`):

- EAX, EDX, ECX, stack

in 64 bits, standard SysV ABI:

- RDI, RSI, RDX, RCX, R8, R9, stack

# Known code?

Interesting open source libraries (version numbers from 6.5 release, Aug 2014):

- BGET: memory allocator (first dev in 1972!)
- NET-SNMP 5.4.2.1 (2008-10-31)
- newlib: libc
- expat 1.95.2: XML parser (2001!)
- libxml2 2.7.7-82143f4 (2010-11-04)
- OpenSSH 6.3 (2013-09-13)
- OpenSSL 1.0.1e (2013-02-11)
- zlib 1.2.3 (2005-07-18)

Blue Coat states that they backport fixes regularly (without necessarily changing the version string).

# Outline

1. Introduction
2. Storage: filesystems and registry
3. Binaries
4. Kernel and OS mechanisms
5. Understanding internals
6. Security mechanisms
7. Conclusion

# Kernel

## The kernel in practice

- kernel access partially abstracted in `libknl_api.so`
- small (~800 KiB), basic primitives:
  - interrupt/exception handling
  - semaphores/locks
  - message passing
  - drivers
- `ds:1014h` points to a “TEB”-like structure

## Some syscalls

```
Nop
Suicide
Enable_event_logging
Register_worker_address
Symbol_address
Processor_voltage
Semaphore_signal_all
Grow_stack
```

# Kernel: syscall

## 32 bits

- `call dword ptr ds:1018h`
- parameters in structure pointed by `eax`

```
kernel_req      struc
    field_0         dd ?
    return_code     dd ?
    return_code2    dd ?
    arg0            dd ?
    arg1            dd ?
    arg2            dd ?
    arg3            dd ?
    sys_num         dd ?
kernel_req      ends
```

## 64 bits

- `call [ds:0FFFFFF800000020h]`
- parameters in structure pointed by `rdi`

```
knl_req64       struc
    field_0         dq ?
    retcode         dq ?
    arg0            dq ?
    arg1            dq ?
    arg2            dq ?
    arg3            dq ?
    sysnum          dq ?
    field_38        dq ?
knl_req64       ends
```

# Memory organization

## Back to the 90s

- protected mode
- **everything** in *ring 0* (mentioned in US7539818 patent ;)

## ELF mapping: at boot, once and for all

```
Unpacking executables...
    Unpacking sequencer.exe elapsed time: 0s, 0ms, 326us
    Unpacking ata.exe elapsed time: 0s, 0ms, 413us
[...]
Relocating executables...
    Relocating sequencer.exe elapsed time: 0s, 2ms, 356us
    Relocating ata.exe elapsed time: 0s, 0ms, 559us
10 executables relocated; total unpack and reloc time 0s, 20ms, 550us
```

# Outline

1. Introduction
2. Storage: filesystems and registry
3. Binaries
4. Kernel and OS mechanisms
5. **Understanding internals**
6. Security mechanisms
7. Conclusion

# Making things easier: our tools

IDA plugins:

- automatically comment function with source filename (from debug logs)
- automatically rename functions from debug log strings
- automated syscall recognition (with syscall name, parameters)
- CLI structures dumper to list all CLI commands

BCFS: FUSE tool to mount system images:

- file access: read/write (without size change)
- CPVE access: read/write
- automatic CRC/HMAC calculation

Tools are internal PoCs and are not going to be released.

# Getting more info: useful tricks

Enable debug info, by modifying BCFS (physical access or RCE needed):

- kernel "*printk*": CPVE 4,0 |= 0x400
- debug mode: set `customer_release` to 0 in `main_cr.cfg`

230+ CLI extensions in debug mode:

- list with ".", access with ".extension"
- examples: `cfg`, `policy`, `cag`, `mgmt`, etc.

## Example commands

- `.mgmt show-adv-urls`
- `.svc ashowstate`
- `.<ext> logaddmask all` then `.<ext> logshow`
- `.policy dbgtraceon`

# CLI extension example

```
ProxySG VA 1818181818>.cag
.cag extension usage
commands:
  logshow       : display contents of the debug log
  logaddmask    : add a mask to the debug log
  logsubmask    : remove a mask from the debug log
  log2console   : toggling logging to the console
  logreset      : reset the CAG debug log
  gzip-allow    : determine if gzip allowed in responses
ProxySG VA 1818181818>enable
Enable Password:
ProxySG VA 1818181818#_
```

# GDB

Kernel includes GDB stub! But finding how to activate it took me weeks :(

- CPVE 4,0 |= 0x8
- multiplexed on COM1 with console
- send 0x18, 0x14 on COM port to activate
- (non-standard) text paging is handled server-side, patch client or use `monitor util height 1000000`

## GDB `monitor` extensions (kernel side)

```
Current debug extensions:
 name knl, Function 0x1261500
 name util, Function 0x1028786E0
 name scorpius, Function 0x1028487E0
```

## Some `knl` extensions

- `processes`: display all active processes.
- `pd`: display the contents of a process descriptor.
- `images`: display details of loaded ELF files.

# Practical understanding: HTTP parsing

Goal: find function for HTTP response parsing

- activate HTTP debug mode at `https://x.x.x.x:8082/HTTP/debug`
- make request through proxy
- get log
- read interesting function name
- look for function in `libhttp.exe.so`

Example log (simplified, most recent first):

```
HTTP CW 95B72F20: Parse_request called. beg=57DE3000 end=57DE30DC length=220
HTTP CW 95B72F20: Parse_request
HTTP CW 95B72F20: Should_tunnel_on_error
HTTP CW 95B72F20: Read_request
HTTP CW 95B72F20 POLICY: Evaluating PE_POLICY_CHECKPOINT_NEW_CONNECTION
HTTP CW 95B72F20: Transaction_startup
HTTP CW 95B72F20: Init_state
```

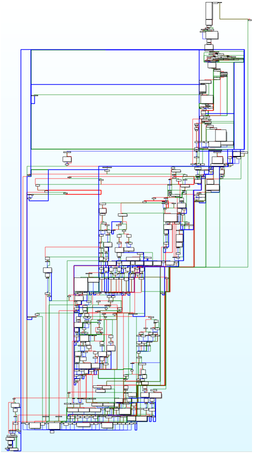
# Going deeper: Hell

Locating the code is the easy part.
Problems:

- **HUGE** functions (16 KiB!, see CFG)
- C++ everywhere
- IDA struggles with calling convention
- threads, everywhere!

Dynamic debugging howto:

- find image base using `monitor knl image libhttp.exe.so` in GDB
- relocate binary in IDA
- set breakpoint in Proxy SG CLI: `conf t; debug; breakpoint-set 0 B X <ADDR>`
- break and connect!

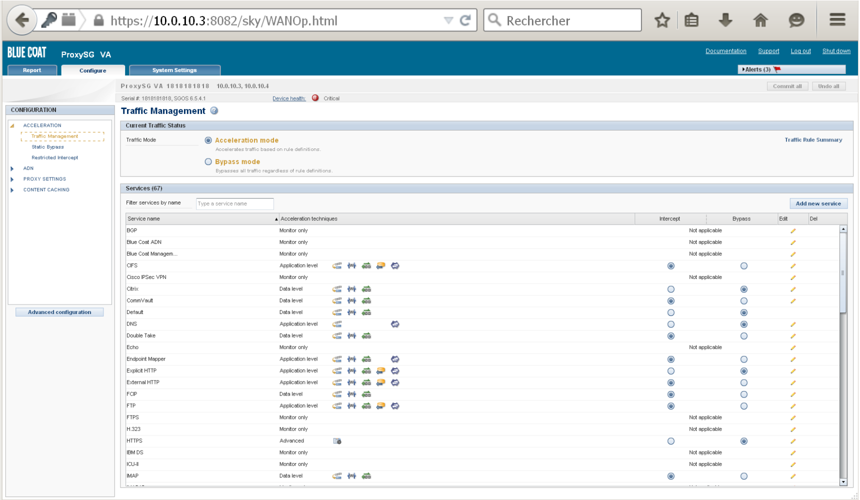


# Outline

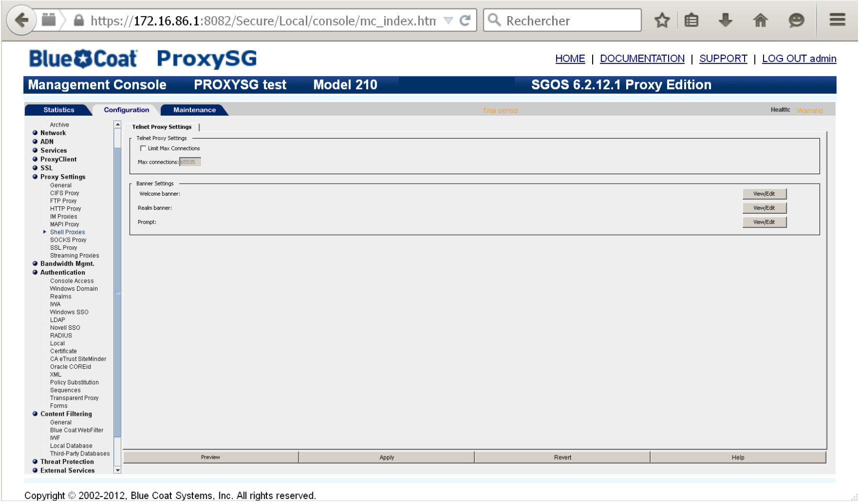
# Application security

- authentication:
  - local passwords are hashed with FreeBSD MD5 crypt ($1$), Blowfish supported
  - dozens of schemes supported: LDAP, AD, etc.
- default protocols: only HTTPS and SSH
- read-only or admin accounts
- OS trust:
  - PKCS7 signed updates (SHA-512/RSA-2048)
  - local images:
    - $<$ 6.5: CRC only
    - $\geq$ 6.5: HMAC SHA-1
- crypto:
  - openssl
  - critical random data is generated securely

# Administration interface (Flash)



# Administration interface (Java)



# Administration interface

- actually *POST*s CLI commands, in an *enable* shell
- restricted commands for *read-only* users
- Java interface specifics:
  - also uses a kind for RPC mechanism (`/Secure/Local/console/pod`)
  - also implements its own HTTPS "client"

## Request (simplified)

```
POST /Secure/Local/console/install_upload_action/cli_post_setup.txt
Host: 10.0.10.3:8082
Authorization: Basic YWRtaW46dGVzdA==
[...]
Cookie: bcsi.logout=0; BCSI_MC=605032960:1

---------------7d518638300904
Content-Disposition: form-data; name="file"

show version
---------------7d518638300904-
```

## Response data

```
ProxySG VA 1818181818#(config)show version
Version: SGOS 6.4.1.2 MACH5 Edition
Release id: 90192
UI Version: 6.4.1.2 Build: 90192
Serial number: XXXXXXXXXX
NIC 0 MAC: 000FF9B6006F
There were 0 errors and 0 warnings
```

# System-level security

## BAD

- no stack canaries
- no ASLR
- everything in ring0
- kernel callgate at a fixed address

## GOOD

- NX enabled on most platforms since 6.2. 300/600 support added in 6.5.7.1 and 6.2.16.3
- BGET heap: asserts check for meta-data coherence (*unlink* attacks impossible)
- read-only FS for binaries makes it (way) harder to backdoor OS
- physical access (or code exec) is needed to change system image as updates are signed

# Exploitability

## Facilitating exploits

- previous slide :)
- vtables everywhere
- only C++ code => more memory corruption bugs (vs script/safer languages)

## Hurdles

- no second chance (*ring0*)
- no ASLR but mapping different for each version
- custom payload needed
- guard pages

# Outline

# Conclusion

## Findings

- unusual, entirely proprietary OS design
- no user/kernel isolation or exploit hardening (historical for performance?)
- no vulnerabilities found (I didn't look for them!). . .
- . . . but Blue Coat release notes document plenty of fixes for "software restarts"

## Recommendations

- use a dedicated (V)LAN for administration
- monitor the event log
- investigate reboots
- physically protect appliances
- use secure passwords (of course!)

# Evolutions in ProxySG

### Security enhancements in recent versions

- NX support for 300/600 added in 6.5.7.1 and 6.2.16.3
- bootchain and system image validation (hashes published by Blue Coat)
- *Secure boot* in pre-release, available in a future release
- debug (GDB, CLI extensions) support removed

We are currently discussing further security enhancements, such as user/supervisor separation, with Blue Coat. Release timing and platform support are still under discussion.

# End

Questions?

**Thanks!**

- Stéphane D. for his work on BCFS and the tikz figures :)
- Stéphane L. and AGI for giving me the opportunity to work on Blue Coat

# Outline

8 Backup slides

# System disk organization (UEFI mode)

## Files on FAT32 partition

`/sgos/boot/systems/diag.si`
`/sgos/boot/systems/system.si`
`/sgos/boot/meta.txt`
`/sgos/fbr.con`
`/EFI/BOOT/BOOTx64.EFI`
`/EFI/BOOT/osloader.si`

*.si* files use BCFS

## New: UEFI

- `BOOTx64.EFI` replaces `starter.si`
- `osloader.si` contains a copy of `BOOTx64.EFI`

## New: `diag`

Linux diagnostic system:

- check hardware health
- interesting `cli` binary, with symbols :)

# Boot sequence (UEFI)

1. UEFI
2. BOOTx64.EFI
3. desired system is selected
4. *prekernel.exe* is started, first file entry in *system.si* FS
5. *prekernel.exe* setups GDT, IDT, etc. and starts *kernel.exe* (2nd entry)
6. kernel starts *sequencer.exe*, (3rd entry)
7. *sequencer.exe* parses the *main_cr.cfg* script
8. *main_cr.cfg* includes *main_common.cfg* which starts everything

Way simpler than BIOS boot.