Know your Unknowns

Techniques for analyzing unknown software

(and dynamic reversing in general)

http://Technologeeks.com/

The Problem

Presented with a new software sample, how do we determine:

- What does it do?

- How does it do it?

- Is it a threat?

http://Technologeeks.com/

Approaches

- Static analysis
  - Requires significant RE skills
  - May be costly and time consuming
  - Binaries may be deliberately obfuscated or use runtime tricks
- Dynamic analysis
  - Often a far simpler approach, no disassembly required
  - Much easier – just fire up and let it run
  - Not without risks of its own

Static Analysis, in a nutshell

- Proper static analysis requires a good disassembler and debugger
- Sometimes you can get lucky with a few simple commands:
  - nm –m (or jtool –S)
  - strings (or jtool –d __TEXT.__cstring)

- Better grade malware likely to be obfuscated:
  - Strings may be “encrypted” (garbled with some key stashed in binary)
  - Symbols may be loaded dynamically via dlopen(3) rather than imports

http://Technologeeks.com/

Dynamic Analysis: Caveats

- Binary may behave differently if detecting inspection or quarantine
- Binary may attempt to thwart invasive inspection
  - Common trick: PTRACE_ATTACH (Linux) or PT_DENY_ATTACH (*OS)
  - Increasingly common trick: llvm obfuscator, jump into syscalls/gadgets
- Binary may run out of control if not properly quarantined

http://Technologeeks.com/

Dynamic Analysis: What to monitor

- What happens inside a process memory stays inside process memory
  - Very hard to track reliably (memory is volatile)
  - You'll usually need a debugger, and probably don't always need to go that deep

- What happens on system-wide objects is of more interest:
  - Files and file system activity
  - Device/hardware access
  - Network sockets

http://Technologeeks.com/

Dynamic Analysis: Techniques*

- Snapshots/Polling
  - Always doable, always undetectable
  - Prone to racing with target process, may miss critical events
- Continuous Process Monitoring
  - Launch binary, keeping close watch on operations using OS tracing APIs
  - Gamut ranges from passive (trace/log only) to full invasive (stop, debug)
  - Possibly detectable by App.
- System Wide Monitoring
  - Inspect overall system performance during target process execution
  - Focus on target process in particular
  - Likely less detectable by App (and can be masked)

*** - Runtime code injection/live debugging techniques intentionally left out scope of this talk.**

Snapshots

- Variety of tools can be used to snapshot:
  - MacOS: sample(1), lsof
  - *OS: procexp (files, regions, threads, core)
  - Linux: /proc/*pid* files (and related tools)
    - Specifically, maps, fd/, fdinfo/, wchan, syscall, to name but a few

http://Technologeeks.com/

Process monitoring: file system activity

- Android (Linux): `inotify(7)`
- Very basic API, provides file/directory notifications
  - Must `inotify_add_watch(2)` for every pathname watched
- DOES NOT provide PID of actor
  - Still useful if system is idle save for tested PID
- Alternative: Polling through /proc/*pid*/fd

http://Technologeeks.com/

Process monitoring: file system activity

- *OS: FSEventsd
  - `/dev/fsevents` character device
  - Caller must be root, `open(2)` device, and use `ioctl(2)`
  - Subsequent `read(2)` operations get system-wide activity notifications
  - Apple provides `FSEventStreamRef` APIs through `CoreFoundation`
    - Use CoreServices' (FSEvents') fseventsd helper daemon as event relay
    - When using fseventsd, don't need root access, but actor information is lost.

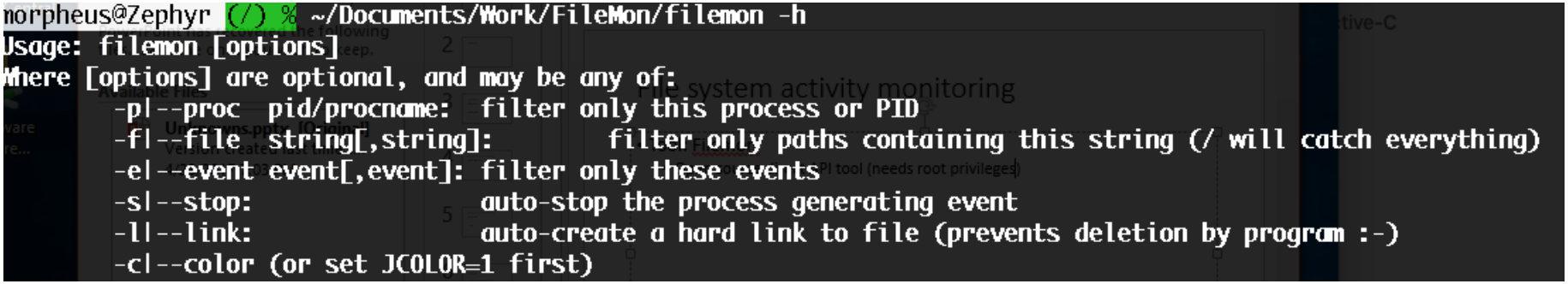
http://Technologeeks.com/

Process monitoring: file system activity

- Tool: filemon
  - Pays homage to Mark Russinovich's awesome tool
  - Displays all file system activity
  - Can (race to) save all temporary files (by auto-hard linking)
  - Can (race to) stop process on file access (by auto kill –STOP)

http://Technologeeks.com/

Demo: filemon



http://Technologeeks.com/

Process Monitoring: Network Activity

- *OS: `com.apple.network.statistics` `PF_SYSTEM` socket
  - Undocumented, described in MOXiI (1st), but significantly beefed up since then
- Used by Apple's `nettop(1)`
  - Unfortunately, a closed source tool ☹
- lsock
  - Fortunately, an open source tool ☺
  - http://NewOSXBook.com/
- No comparable notification based mechanism on Linux/Android.

http://Technologeeks.com/

Process Monitoring: Linux/Android

- Standard Tool: strace(1): Trace system calls
  - Built-in (Linux) or available via emulator/compilation (Android)
  - Expensive – but manageable - overhead of x4 on average
  - Traces all system calls in C-style APIs, including:
    - Timestamps (-t[t[t]], and –T, for time in call)
    - Instruction pointer at time of call (-i)
    - Verbose arguments (-v[v])
    - Following children processes or threads (-f)
- Semi-standard ltrace(1) can trace library calls
  - Downloadable (Linux) or compilable (Android)
  - Ridiculously expensive (x20-x100!), uses PTRACE_SINGLESTEP

http://Technologeeks.com/

Process Monitoring: Linux/Android

- Both strace(1) and ltrace(1) are passive – cannot perform any hooks

- Non-standard tool: jtrace
  - http://NewAndroidBook.com/tools/jtrace.html

- By itself, an improved version of strace(1) providing:
  - Proper ARM64 support, as well as x86_64 and ARMv7
  - Android-aware syscall argument support (binder, input messages, etc)
  - Colors

Process Monitoring: Linux/Android

- Most powerful feature of jtrace: **Plugins**

- Provide your own callbacks in an .so, load it into jtrace, and it will:
  - … call you back on your syscalls of interest
  - … provide a simple, **architecture agnostic API** for you to get args and memory
  - … allow you to also manipulate system call, before and after execution

- All this, **without** any code injection into target process

http://Technologeeks.com/

Process Monitoring: MacOS/iOS

- MacOS and iOS already have the world's most powerful sandbox
- Idea: Why not harness the sandbox for tracing?

```
morpheus@Zephyr (~) %cat /tmp/trace.sb
(version 1)
(trace "/tmp/tracing.out")
morpheus@Zephyr (~) %# Run some command, under sandbox profile:
morpheus@Zephyr (~) %sandbox-exec -f /tmp/trace.sb ls /
Applications                Users                       etc                         net
Library                     Volumes                     home                        private
Network                     bin                         installer.failurerequests   sbin
PanicDumps                  cores                       mnt                         tmp
System                      dev                         mnt1                        usr
morpheus@Zephyr (~) %cat /tmp/tracing.out
(version 1) ; Thu Apr 27 22:14:16 2017
(allow process-exec* (path "/bin/ls"))
(allow process-exec* (path "/bin/ls"))
(allow file-read-metadata (path "/usr/lib/dyld"))
(allow file-read-metadata (path "/usr/lib/dyld"))
(allow file-read-metadata (path "/usr/lib/libutil.dylib"))
```

Invasive Process Monitoring

- Nothing beats living inside the process – code injection
- Dynamically linked binaries are open to suggestion…
  - MacOS/*OS: DYLD_INSERT_LIBRARIES
  - Linux/Android: LD_PRELOAD
  - All systems: runtime code injection*
- Caveat: highly invasive, may undermine target stability if not careful

*** - Runtime code injection/live debugging techniques intentionally left out scope of this talk.**

Invasive Process Monitoring

```c
#include <fcntl.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

// This is the expected interpose structure
typedef struct interpose_s {
    void *new_func;
    void *orig_func;
} interpose_t;

int my_open(const char *path, int oflag, ...)
{
        int rc = open (path,oflag);
        printf("Opened %s at FD %d\n", path, rc);
        return (rc);
}

int my_connect(int socket, const struct sockaddr *address, socklen_t address_len)
{
        struct sockaddr_in *sin = (struct sockaddr_in *) address;
        if (sin->sin_family == AF_SYSTEM) {
                printf("connect on system socket\n");
                }
        else
        printf("Intercepted connect to AF %d, %s:%d\n", sin->sin_family, inet_ntoa(sin->sin_addr),ntohs(sin->sin_port));
        int rc = connect(socket, address, address_len);
        return (rc);
}

static const interpose_t interposing_functions[] \
    __attribute__ ((used,section("__DATA, __interpose"))) = {
        { (void *)my_open,  (void *) open },
        { (void *)my_connect,  (void *) connect },
    };
```

http://Technologeeks.com/

System-wide activity monitoring

- Linux/Android: ftrace
- MacOS: dtrace
- *OS: kdebug

http://Technologeeks.com/

System-wide activity monitoring: Linux ftrace

- Linux's most under-documented feature

- **COMPLETE** tracing of all kernel operations – functions or subsystems
  - Kernel tracing is %#$%#% unbelievable, but out of scope
  - For the scope of our discussion, syscalls or raw_syscalls subsystems suffice

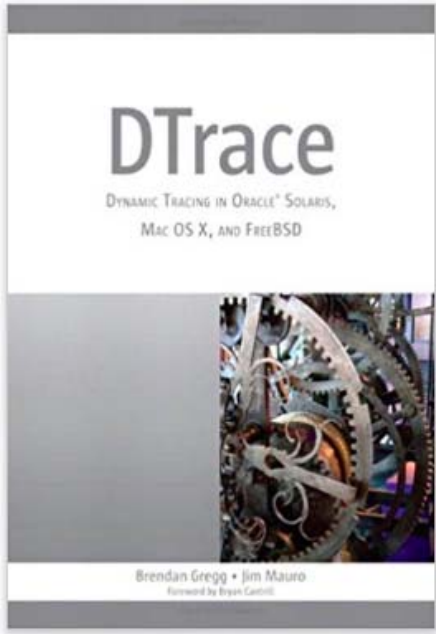
- Exact scope and subsystems depend on compiled kernel options..
  - .. But you can always recompile the kernel to suit your needs
  - You can get even more granular by compiling your own kernel module

http://Technologeeks.com/

System-wide activity monitoring: MacOS dtrace

- Most powerful tracing API of any OS, period
  - Unfortunately (but understandably) not provided by AAPL in iOS variants
  - Further impaired in MacOS 10.11+ due to SIP (but still usable)
- Provides a complete *language* for constructing “probes”
  - Superb reference: PH Dtrace book
- Probes fire when conditions met
- Probe may read/write/modify target
- Caveat: Expensive.

http://Technologeeks.com/



Demo: dtrace

- Get all syscalls by all processes with one line of code:
  - dtrace -n 'syscall:::entry { printf("%d:%s", pid, execname); }'

- Limit to open(2) and friends, show filename:
  - dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'

http://Technologeeks.com/

System-wide activity monitoring: kdebug

- Available in all of Apple's OSes (but requires root)
- Provides a firehose of information on system activity
- Apple's own tools:
  - `latency`: monitors scheduling and interrupt latency
  - `iprofiler`: Xcode (instrument) profiler
  - `sc_usage`: show system call usage statistics
  - `fs_usage`: report file system activity syscalls
  - `trace`: generic utility for kdebug facility
- kdebugView (http://NewOSXBook.com/tools/kdv.html)
- Use it or lose it (before AAPL slaps an entitlement on this as well..)

http://Technologeeks.com/

Links, etc:

- Much more on the *OS tools coming in MOXiI II, Vol I (July 2017!)
- Tools available at http://NewOSXBook.com/ (*OS)
  http://NewAndroidBook.com/ (Android)
- Check out http://Technologeeks.com/ for:
  - MacOS/iOS Internals
  - Android Internals