LEARN HOW TO BUILD YOUR OWN UTILITY TO MONITOR MALICIOUS BEHAVIORS OF MALWARE ON MACOS

Kai Lu(@k3vinlusec), Senior Security Researcher from Fortinet's FortiGuard Labs
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowledge
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowledgement
BIO

• Security Researcher
• Malware and vulnerability research/analysis on macOS and Android.
• Finding vulnerability, vulnerability analysis and found more than 30 vulnerabilities in popular products from Microsoft, Google, Apple, Adobe, etc.
• Top 100 Security Researcher ranked by Microsoft in 2016
• Speaker at ToorCon 19 San Diego
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowledgment
Introduction to FortiAppMonitor

*FortiAppMonitor* is a powerful utility to monitor kinds of behaviors of program on macOS. As a malware analyst or security researcher, having a powerful dynamic analysis utility is vital to be effective and efficient. This utility enables us to understand malware capabilities and quickly analyze the malicious behaviors of malware targeting on macOS. The capabilities of the utility cover the following features.

1. Monitor process execution with command line arguments and process exit.
2. Monitor all common file system events, including file open, read, write, delete, and rename operations.
3. Monitor network activities, including UDP, TCP, DNS query and response, ICMP for both IPv4 and IPv6 protocols.
4. Monitor dylib loading event.
5. Monitor KEXT loading and unloading events.
Introduction to FortiAppMonitor

- Supported OS: macOS 10.11, 10.12, 10.13, 10.14 (Beta)
Introduction to FortiAppMonitor

- It provides a fine-grained filter.
Introduction to FortiAppMonitor

- It also provides a powerful search functionality.
Introduction to FortiAppMonitor

- Save all records as a JSON format file.
- Copy record data in GUI using shortcut key “Command+C” to clipboard.
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowlege
Implementation to FortiAppMonitor
(GUI design and implementation)

- **Electron + Node.js + W2UI**

  - **Electron** is a framework for creating native applications with web technologies like JavaScript, HTML, and CSS. It takes care of the hard parts so you can focus on the core of your application. Node.js Addons are dynamically-linked shared objects, written in C++, that can be loaded into Node.js using the `require()` function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries. All core of the tool in user space is developed by Node.js Addons.

- **W2UI** is a small JavaScript UI library with a complete set of widgets: layout, grid, sidebar, toolbar, tabs, fields, popup, utilities.
Implementation to FortiAppMonitor (GUI design and implementation)

Control Buttons

Search and Save

Filter options

Grid Panel

Record Details

```
{
    "recid": 11,
    "time": "1532989529.128313431",
    "eventType": "Network(Outbound TCP)",
    "process": "epctrl",
    "pid": 564,
    "user": "root",
    "message": "Connect to remote host 172.30.194.254:8013",
    "extra": "{"parent process":"launchd","ppid":1,"uid":0}"
}
```
Implementation to FortiAppMonitor

- Memory usage and performance

<table>
<thead>
<tr>
<th>Record Lines</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>87,218</td>
<td>~283M</td>
</tr>
<tr>
<td>193,664</td>
<td>~423M</td>
</tr>
<tr>
<td>267,064</td>
<td>~511M</td>
</tr>
<tr>
<td>297,298</td>
<td>~637M</td>
</tr>
<tr>
<td>317,195</td>
<td>~682M</td>
</tr>
</tbody>
</table>
Implementation to FortiAppMonitor

• Workflow and Design
Common malicious behaviors of malwares targeting on macOS

1. File system activities, such as file open, read, write, rename and delete.
2. Network activities, such as udp, tcp, dns query and response, etc.
3. Process creation and exit
4. Dynamic library .dylib loading event
5. Load malicious kernel extensions (i.e. rootkits)
6. ......
What capabilities does the utility offer?

1. Monitor process execution with command line arguments and process exit
2. Monitor network activities
3. Monitor all common file system events
4. Monitor dylib loading event
5. Monitor KEXT loading and unloading events
Apple introduces KAuth with macOS 10.4, as a new KPI to allow third parties – primarily security software developers – to intercept select operations. As a KPI, any such third party must reside in kernel mode.

The Mandatory Access Control Framework - commonly referred to as MACF - is the substrate on top of which all of Apple’s securities, both macOS and iOS, are implemented.

The MACF offers the more detailed granularity than KAuth. It’s perfectly suited for third party security products. It’s a very good choice to implement monitoring kinds of events.

In this utility, I use MACF to monitor process execution, file system events, dylib loading event, KEXT loading and unloading events.

Use Socket Filters to monitor network activities.

Use kqueue(a kernel event notification mechanism) to monitor process exit.
Implementration to FortiAppMonitor
(Monitor process execution using MACF)

The `mac_policy_ops` structure includes more than 300 policy module operations. You only need to initialize the operations you concern.

```c
int mac_policy_register(struct mac_policy_conf *mpc,
                        mac_policy_handle_t *handlep, void **xd);

struct mac_policy_conf {
    const char     *mpc_name;  /* policy name */
    const char     *mpc_fullname;  /* full name */
    const char     **mpc_labelnames;  /* managed label namespaces */
    unsigned int   mpc_labelname_count;  /* number of managed label namespaces */

    struct mac_policy_ops  *mpc_ops;  /* operation vector */
    int                  mpc_loadtime_flags;  /* load time flags */
    int                  *mpc_field_off;  /* label slot */
    int                  mpc_runtime_flags;  /* run time flags */
    mpc_t                mpc_list;  /* List reference */

    void static struct mac_policy_ops mmPolicyOps = {
        .mpo_cred_label_update_execve = processExecWithArgsHook,  //hook process execution, including arguments
        .mpo_vnode_check_open = openFileHook,  //hook opening or creating a file for reading or writing
        .mpo_vnode_check_unlink = deleteFileHook,  //hook file deletion
        .mpo_vnode_check_read = readFileHook,  //hook file read
        .mpo_vnode_check_rename = renameFileHook,  //hook file rename
        .mpo_vnode_check_write = writeFileHook,  //hook file write
        .mpo_file_check_mmap = dylibLoadHook,  //hook dylib load event
        .mpo_kext_check_load = kextLoadHook,  //hook kext load event
        .mpo_kext_check_unload = kextUnloadHook  //hook kext unload event
    };
};
```
Implementation to FortiAppMonitor
(Monitor process execution using MACF)

• Use the operation `mpo_cred_label_update_execve_t` to monitor process execution with all command line arguments.

```c
typedef int mpo_cred_label_update_execve_t(
    kauth_cred_t old_cred,
    kauth_cred_t new_cred,
    struct proc *p,
    struct vnode *vp,
    off_t offset,
    struct vnode *scriptvp,
    struct label *vnode_label,
    struct label *scriptvnode_label,
    struct label *execlabel,
    u_int *csflags,
    void *macpolicyattr,
    size_t macpolicyattrlen,
    int *disjointp
);
```

@brief Update credential at exec time
@param old_cred Existing subject credential
@param new_cred New subject credential to be labeled
@param p Object process.
@param vp File being executed
@param offset Offset of binary within file being executed
@param scriptvp Script being executed by interpreter, if any.
@param vnode_label Label corresponding to vp
@param scriptvnode_label Script vnode label
@param execlabel Userspace provided execution label
@param csflags Code signing flags to be set after exec
@param macpolicyattr MAC policy-specific spawn attribute data.
@param macpolicyattrlen Length of policy-specific spawn attribute data.
@see mpo_execve
@see mpo_cred_check_label_update_execve_t
@see mpo_vnode_check_exec_t
Implemetation to FortiAppMonitor
(Monitor process execution using MACF)

- The callback processExecWithArgsHook

The parameter csflags is code signing flags to be set after execution. It is a key parameter for getting the command line arguments taken by process execution. We can calculate the memory address of an object to image_params by subtracting the offset of ip_csflags in the image_params structure from the memory address of csflags.
Implenentation to FortiAppMonitor
(Monitor process execution using MACF)

- The image_params structure is defined in bsd/sys/imgact.h, which acts as a container for passing around program parameters between functions called by execve().
- The member variable ip_csflags of the image_params structure is actually the parameter csflags taken by the callback processExecWithArgsHook. Once we have the address of image_params structure pointer, we can reference its member variables ip_startargv and ip_endargv to get the data buffer for the command line arguments of process execution.
- **Note that** the member variable ip_argc represents the amount of the command line arguments. It is also worth noting that each parameter in buffer is separated with '\0', so you need to do a string replacement operation to get the complete string of all command line arguments.
Implementation to FortiAppMonitor
(Monitor process execution using MACF)

• Demo of monitoring process execution
Implementetation to FortiAppMonitor  
(Monitor file system events using MACF)

• It can monitor all common file system events, including file open, read, write, rename, and delete operations.

• The operations in the mac_policy_ops structure related to file system events are listed below.
  1. mpo_vnode_check_open_t *mpo_vnode_check_open;  //File open
  2. mpo_vnode_check_read_t *mpo_vnode_check_read;  //File read
  3. mpo_vnode_check_unlink_t *mpo_vnode_check_unlink;  //File delete
  4. mpo_vnode_check_write_t *mpo_vnode_check_write;  //File write
  5. mpo_vnode_check_rename_t *mpo_vnode_check_rename;  //File rename
Implementation to FortiAppMonitor
(Monitor file system events using MACF)

• Divide the five callbacks of file operations into two categories depending on their implementation.
  1. File open, read, write, and delete operations
  2. File rename operation.

• In the 1st category, pick file delete operation to use as an example for showing technical details. The others operations in this category are similar to the callback for the file delete operation.
Implementation to FortiAppMonitor (Monitor file system events using MACF)

- The callback of the file delete operation is `mpo_vnode_check_unlink`.

```c
typedef int mpo_vnode_check_unlink_t(
    kauth_cred_t cred,
    struct vnode *dvp,
    struct label *dlabel,
    struct vnode *vp,
    struct label *label,
    struct componentname *cnp
);

static int deleteFileHook(kauth_cred_t cred,
    struct vnode *dvp,
    struct label *dlabel,
    struct vnode *vp,
    struct label *label,
    struct componentname *cnp)
{
    errno_t status = KERN_FAILURE;
    clock_sec_t secs=0;
    clock_nsec_t nanosecs=0;
    clock_get_calendar_nanosix(&secs, &nanosecs);

    const char *vname = NULL;
    char path[MAXPATHLEN]=0;
    int pathlen = MAXPATHLEN;
    char procName[256]=0;
    char procPathName[256]=0;

    uid_t uid = -1;
    pid_t pid = -1;
    pid_t pid = -1;

    //get name
    vname = vnode_getname(vp);
    //get path
    vn_getpath(vp, path, &pathlen);
    //null-terminate, just to be safe
    path[MAXPATHLEN-1] = 0;
    //get uid
    uid = kauth_getuid();
    //get pid
    pid = proc_selfpid();
    //get ppid
    ppid = proc_selfppid();
    //get process name
    proc_selfname(procname, 266);
    //get parent process name
    proc_name(ppid, procPathName, 256);

    @brief Access control check for deleting vnode
    @param cred Subject credential
    @param dvp Parent directory vnode
    @param dlabel Policy label for dvp
    @param vp Object vnode to delete
    @param label Policy label for vp
    @param cnp Component name for vp
    @see mpo_check_rename_to_t
```
Implemetation to FortiAppMonitor
(Monitor file system events using MACF)

• The callback of file rename operation is mpo_vnode_check_rename_t.
• It’s a little different from the callbacks of other file operations. In this callback, we need to get both the original path and destination path of the file.

```c
typedef int mpo_vnode_check_rename_t(
    kauth_cred_t cred,
    struct vnode *dvp,
    struct label *dlabel,
    struct vnode *vp,
    struct label *label,
    struct componentname *cnp,
    struct vnode *tdvp,
    struct label *tdlabel,
    struct vnode *tvp,
    struct label *tlabel,
    struct componentname *tcnp
);
```
Implement to FortiAppMonitor
(Monitor file system events using MACF)

• How to get the full original path of the file to be renamed?

```c
origvname = vnode_getname(vp);
vn_getpath(dvp, origpath, &origpathLen);
strncat(origpath, "/", 1);
strncat(origpath, origvname, strlen(origvname));
origpath[MAXPATHLEN-1] = 0;
```

• How to get the full destination path of the file to be renamed?

• Note: At the beginning, I also used this same way to get the destination path. However, sometimes the variable tvp is NULL, so then I get the destination vnode name by tcnp->cn_nameptr instead of vnode_getname(tvp).

```c
//sometimes, tvp is NULL, so get the destination vnode name by tcnp->cn_nameptr instead of vnode_getname(tvp).
vn_getpath(dvp, dstpath, &dstpathLen);
strncat(dstpath, "/", 1);
strncat(dstpath, tcnp->cn_nameptr, strlen(tcnp->cn_nameptr));
dstpath[MAXPATHLEN-1]=0;
```
Implementation to FortiAppMonitor
(Monitor file system events using MACF)

• The variable tcnp is a pointer to the componentname structure.
Implementation to FortiAppMonitor
(Monitor file system events using MACF)

• Demo of monitoring file system events
Implementation to FortiAppMonitor
(Monitor dylib loading event using MACF)

• The dynamic loader, dyld is designed to dynamically link and load the dynamic libraries on the macOS platform. The dynamic library has the extension .dylib, which is similar to DLL files on the Windows operating system, and the .so files on the Linux operating system.

• In order to monitor a dylib loading event, we first need to understand how the dyld dynamic loader works inside. The dyld is open sourced by Apple at https://opensource.apple.com/source/dyld/dyld-519.2.2/.
Implementaton to FortiAppMonitor (Monitor dylib loading event using MACF)

• Here, I only provide the key analysis of code. The following is the code snippet of the function mapSegments of the class ImageLoaderMachO.

```c
void ImageLoaderMachO::mapSegments(int fd, uint64_t offsetInFat, uint64_t lenInFat, uint64_t fileLen, const LinkContext* context)
{
    // find address range for image
    inptr_t slide = this->assignedSegmentAddresses(context);
    if (context.verboseMapping) {
        if (offsetInFat != 0)
            dyld::log("dyld: Mapping %lu\n", this->getFatPath());
    }
    // map in all segments
    for(unsigned int i = 0; i < e.segments.size(); i++) {
        vn_offset_t fileOffset = vn_offset_t{(uint64_t)segments[i].size};
        vn_size_t size = segments[i].size;
        uintptr_t requestedLoadAddress = segPreferredLoadAddress(i) + slide;
        int protection = 0;
        if (segments[i].isExecutable()) { // If has text-reloc, don't set x-bit initially,
            // Instead set it later after text-reloc have been done.
            if (segments[i].isExecutable() && (segments[i].isTextEnd() && (slide != 0)))
                protection |= PROT_EXEC;
            if (segments[i].isReadable())
                protection |= PROT_READ;
            if (segments[i].isWritable())
                protection |= PROT_WRITE;
            // rdar://problem/22525618 force __LINKEDIT to always be mapped read-only
            if (strcmp(segments[i].filename, "__LINKEDIT") == 0)
                protection |= PROT_READ;
        }
        uint8_t protection = (uint8_t)protection;
        // Initially map __IMPORT segments R/W so dyld can update them
        if (segments[i].isSymbolicImport())
            protection |= PROT_WRITE;
        // wholly zero-fill segments have nothing to mmap() in
        if (protection == 0) {
            if (fileOffset + size > fileLen) {
                dyld::log("truncated mach-o error: segment extends to 11u which is past end of file\n\tsegment: %u, (uint64_t) fileOffset + size, fileLen);"
            }
        }
        void* loadAddress = mmap(void*, requestedLoadAddress, size, protection, MAP_FIXED | MAP_PRIVATE, fd, fileOffset);
        if (loadAddress == NULL) {
            dyld::log("mmap\nerror\n\tfd: %d,\n\taddr: %p,\n\tsize: %d,\n\tprot: 0x%x,\n\tflags: MAP_PRIVATE,\n\toff: %u\n", fd, loadAddress, size, protection, fileOffset);
        }
    }
}
```

It maps the segments in Mach-O binary into virtual memory. It uses the function `mmap`, which is a wrapper of the function `mmap` used to perform the mapping. The manual page of the function `mmap` can be found at https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/mmap.2.html.
Implementation to FortiAppMonitor
(Monitor dylib loading event using MACF)

• Because the dylib is an executable program, we only focus on the segment with PROT_EXEC privilege. The parameter flags in the function mmap is taken with MAP_FIXED | MAP_PRIVATE.

• In the mac_policy_ops structure, the module operation mpo_file_check_mmap is used to check the access control for mapping a file. We only filter the mapping with PROT_EXEC privilege and MAP_FIXED | MAP_PRIVATE flags.

```c
typedef int mpo_file_check_mmap_t(
    kauth_cred_t cred,
    struct fileglob *fg,
    struct label *label,
    int prot,
    int flags,
    uint64_t file_pos,
    int *maxprot
);
```

@brief Access control check for mapping a file
@param cred Subject credential
@param fg fileglob representing file to map
@param label Policy label associated with vp
@param prot mmap protections; see mmap(2)
@param flags Type of mapped object; see mmap(2)
@param maxprot Maximum rights
The fileglob structure represents the file to map.

```c
struct fileglob {
    LIST_ENTRY(fileglob) f_meglist; /* list of active files */
    int32_t fg_flag;       /* see fontil.h */
    int32_t fg_count;      /* reference count */
    int32_t fg_messagecount; /* references from message queue */
    int32_t fg_lflags;     /* file global flags */
    kauth_cred_t fg_cred; /* credentials associated with descriptor */
} const struct fileops {
    file_type_t f_type; /* descriptor_type */
    int (*fo_read) (struct fileproc *fp, struct uio *uio, int flags, vfs_context_t cx);
    int (*fo_write) (struct fileproc *fp, struct uio *uio, int flags, vfs_context_t cx);
    int (*fo_ioctl) (struct fileproc *fp, u_long com, caddr_t data, vfs_context_t cx);
    int (*fo_select) (struct fileproc *fp, int which, void *wl, vfs_context_t cx);
    int (*fo_close) (struct fileglob *fg, vfs_context_t cx);
    int (*fo_ioctl) (struct fileproc *fp, struct klopn *kn, struct kvnt_internal_s *kev, vfs_context_t cx);
    int (*fo_drain) (struct fileproc *fp, vfs_context_t cx);
} *fg_ops;
```

We need to find the vnode data structure inside it. The member variable `fg_data` represents either a vnode or socket or SHM or semaphore, depending on the descriptor type. The member variable `fo_type` of the fileops structure simply represents the descriptor type. We only filter the file type, which represents that `fo_type` is equal to `DTYPE_VNODE`.

```c
typedef enum {
    DTYPE_VNODE = 1, /* file */
    DTYPE_SOCKET, /* communications endpoint */
    DTYPE_PSHMEM, /* POSIX Shared memory */
    DTYPE_PSHMEM, /* POSIX Semaphores */
    DTYPE_RSEQ, /* kqueue */
    DTYPE_PIPE, /* pipe */
    DTYPE_PSERVENTS, /* events */
    DTYPE_AVALK, /* obsolete */
    DTYPE_NETPOLICY, /* networking policy */
} file_type_t;
```

At this point, the variable `fg_data` represents the vnode. Once we have the vnode, we’re able to get the path by invoking the `vn_getpath` function.
The implementation of callback for monitoring the dylib loading event.

The 4th parameter prot, and the 5th parameter flags are from the system call mmap. The expression `((prot & 0x4) && (flags & 0x12))` is used to filter the mapping with PROT_EXEC privilege and MAP_FIXED | MAP_PRIVATE flags. The 2nd parameter is a pointer to the fileglob structure, which is defined in the header file bsd/sys/file_internal.h. Since the fileglob structure is a private type, the head file file_internal.h cannot be imported in XCode. Therefore, we need to calculate the corresponding offset of the member variables in the fileglob structure, rather than directly getting the values through fg->fg_data, fg->fg_cred, and fg->fg_ops->fo_type in XCode.
Implementation to FortiAppMonitor
(Monitor dylib loading event using MACF)

• Demo of monitoring dylib loading event
Implementetation to FortiAppMonitor
(Monitor KEXT loading and unloading events using MACF)

• The operation `mpo_kext_check_load_t` in the `mac_policy_ops` structure is used to monitor kext loading event.
• The operation `mpo_kext_check_unload_t` in the `mac_policy_ops` structure is used to monitor kext unloading event.

```c
/*
 * @brief Access control check for kext loading
 * @param cred Subject credential
 * @param identifier Kext identifier
 *
 * Determine whether the subject identified by the credential can load the specified kext.
 *
 * @return Return 0 if access is granted, otherwise an appropriate value for errno should be returned. Suggested failure: EPERM for lack of privilege.
 */
typedef int mpo_kext_check_load_t(
    kauth_cred_t cred,
    const char *identifier
);

/*
 * @brief Access control check for kext unloading
 * @param cred Subject credential
 * @param identifier Kext identifier
 *
 * Determine whether the subject identified by the credential can unload the specified kext.
 *
 * @return Return 0 if access is granted, otherwise an appropriate value for errno should be returned. Suggested failure: EPERM for lack of privilege.
 */
typedef int mpo_kext_check_unload_t(
    kauth_cred_t cred,
    const char *identifier
);
```

The parameter identifier is the kext identifier like “com.xxxxxx.xxxxxx”.
Implementation to FortiAppMonitor
(Monitor KEXT loading and unloading events using MACF)

• Demo of monitoring KEXT loading and unloading events
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• Generally, malware on macOS performs a number of network activities to either retrieve an attack payload from a remote server, or to send sensitive information collected from an infected system to a remote server.

• Monitor network activities (another significant behavior for malware) using Socket Filters (a part of the Network Kernel Extension) on macOS.

• It includes monitoring UDP, TCP, ICMP, DNS query and response data for both IPv4 and IPv6 protocols.
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• The socket filter is a powerful mechanism that enables the interception of network and IPC traffic in the kernel’s socket layer. A socket filter is a filter associated with a particular socket, as shown below.
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

- Network Kernel Extensions can filter inbound or outbound traffic on a socket. They can also filter out-of-band communications, including calls to setsockopt, getsockopt, ioctl, connect, listen, and bind functions.
- The life cycle of a socket filter can be summed up as follows.
  1. Socket filters are installed in the kernel by invoking the sflt_register, typically from the filter’s initialization routine.
  2. Later, when the filter is instantiated on a socket, the protocol calls the filter’s sf_attach_func callback. This callback may return a unique cookie through its first parameter that can be used for tracking storage specific to a given filter instance (attached to a specific socket).
  3. When the filter is detached, the filter’s sf_detach_func callback is invoked. At this point, the filter should free any socket-specific resources that it has allocated (generally in the sf_attach_func).
  4. The socket filter may, at some point, decide that it wishes to be unloaded. If so, it should invoke sflt_unregister. This will prevent the filter from being attached to new sockets in the future and will begin the process of detaching the filter from existing sockets.
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

- **sflt_register**

- The 1st parameter is a pointer to the sflt_filter structure. A socket filter is registered by filling out the desired callbacks in sflt_filter structure.

```c
extern erro_t sflt_register(const struct sflt_filter *filter, int domain, int type, int protocol);
```
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• Currently, It supports 8 socket filters, as seen in the following Table.

<table>
<thead>
<tr>
<th>NO.</th>
<th>Domain</th>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>PF_INET</td>
<td>SOCK_STREAM</td>
<td>IPPROTO_TCP</td>
</tr>
<tr>
<td>2.</td>
<td>PF_INET</td>
<td>SOCK_DGRAM</td>
<td>IPPROTO_UDP</td>
</tr>
<tr>
<td>3.</td>
<td>PF_INET</td>
<td>SOCK_RAW</td>
<td>IPPROTO_ICMP</td>
</tr>
<tr>
<td>4.</td>
<td>PF_INET</td>
<td>SOCK_DGRAM</td>
<td>IPPROTO_ICMP</td>
</tr>
<tr>
<td>5.</td>
<td>PF_INET6</td>
<td>SOCK_STREAM</td>
<td>IPPROTO_TCP</td>
</tr>
<tr>
<td>6.</td>
<td>PF_INET6</td>
<td>SOCK_DGRAM</td>
<td>IPPROTO_UDP</td>
</tr>
<tr>
<td>7.</td>
<td>PF_INET6</td>
<td>SOCK_RAW</td>
<td>IPPROTO_ICMPV6</td>
</tr>
<tr>
<td>8.</td>
<td>PF_INET6</td>
<td>SOCK_DGRAM</td>
<td>IPPROTO_ICMPV6</td>
</tr>
</tbody>
</table>
Implemetation to FortiAppMonitor (Monitor network activities using Socket Filters)

• `errno_t monitorAgent_sflt_attach(void **cookie, socket_t so)`
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• The initializations of the sflt_filter structure. They are, respectively, intended to filter TCP protocol for IPv4 and IPv6.

```c
static struct sflt_filter gSflt_TCPIPv4 = {
    MACOMMONITORAGENT_SFLT_TCPIPv4_HANDLE, 0x01,
    MONITORAGENT_BUNDLE_ID,
    monitorAgent_sflt_unregistered,
    monitorAgent_sflt_attach,
    monitorAgent_sflt_detach,
    NULL,
    NULL,
    NULL,
    NULL, //monitorAgent_sflt_data_in,
    NULL, //monitorAgent_sflt_data_out,
    monitorAgent_sflt_connect_in,
    monitorAgent_sflt_connect_out,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

```c
static struct sflt_filter gSflt_TCPIPv6 = {
    MACOMMONITORAGENT_SFLT_TCPIPv6_HANDLE, 0x01,
    MONITORAGENT_BUNDLE_ID,
    monitorAgent_sflt_unregistered,
    monitorAgent_sflt_attach,
    monitorAgent_sflt_detach,
    NULL,
    NULL,
    NULL,
    NULL, //monitorAgent_sflt_data_in,
    NULL, //monitorAgent_sflt_data_out,
    monitorAgent_sflt_connect_in,
    monitorAgent_sflt_connect_out,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

Only set two callbacks – `sf_connect_in` and `sf_connect_out`. The callback `monitorAgent_sflt_connect_out` is used to intercept calls to the `connect()` system call for outgoing connections. The callback `sf_connect_in` is called by a protocol handler just before a new connection is established for incoming connections.
Implemetation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• The key code snippet of callback monitorAgent_sflt_connect_out.

```c
if(domain == AF_INET) {
    addr = (struct sockaddr_in *)to;
    inet_ntop(AF_INET, &addr->sin_addr, (char*)addrstr, sizeof(addrstr));
    port = ntohs(addr->sin_port);

    protocol = *(uint32_t *)(cookie+0x410);
    pid = *(uint32_t *)(cookie+0x400);
    ppid = *(uint32_t *)(cookie+0x404);
    proiname = (char *)cookie;
}
```

An outgoing connection to a remote IP address and port. For example, when a malware tries to connect to the remote control server on TCP, the callback monitorAgent_sflt_connect_out can record the info of an outgoing connection.

```c
if(domain == AF_INET6) {
    addr6 = (struct sockaddr_in6 *)to;
    inet_ntop(AF_INET6, &addr6->sin6_addr, (char*)addrstr, sizeof(addrstr));
    port = ntohs(addr6->sin6_port);

    protocol = *(uint32_t *)(cookie+0x410);
    pid = *(uint32_t *)(cookie+0x400);
    ppid = *(uint32_t *)(cookie+0x404);
    proiname = (char *)cookie;
}
```
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• The key code snippet of callback `monitorAgent_sflt_connect_in`.

```c
if(cookie != NULL && so != NULL && from != NULL )
{
    domain = *(uint32_t *)(cookie+0x408);

    if(domain == AF_INET)
    {
        addr = (struct sockaddr *)from;
        inet_ntop(AF_INET, &addr->sin_addr, (char*)addrstr, sizeof(addrstr));
        port = ntohs(addr->sin_port);

        protocol = *(uint32_t *)(cookie+0x410);
        pid = *(uint32_t *)(cookie+0x408);
        ppid = *(uint32_t *)(cookie+0x404);
        procname = (char *)cookie;

        if(domain == AF_INET6)
        {
            addr6 = (struct sockaddr_in6 *)from;
            inet_ntop(AF_INET6, &addr6->sin6_addr, (char*)addrstr, sizeof(addrstr));
            port = ntohs(addr6->sin6_port);

            protocol = *(uint32_t *)(cookie+0x410);
            pid = *(uint32_t *)(cookie+0x408);
            ppid = *(uint32_t *)(cookie+0x404);
            procname = (char *)cookie;
        }
    }
}
```

An incoming connection from a remote IP address and port. Similarly, the callback `monitorAgent_sflt_connect_in` can record the incoming connection on TCP.
Implemetation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• To filter UDP, we only set two callbacks – sf_data_out and sf_data_in, respectively, with monitorAgent_sflt_data_out and monitorAgent_sflt_data_in. Others are set as NULL. They allow the interception of incoming and outgoing packets. At this point, we can use them to intercept DNS query and response data packets.
Implemetation to FortiAppMonitor (Monitor network activities using Socket Filters)

• Intercept DNS query

```c
mbuf_copydata(memBuf, 0, querySize, (void*)dnsQuery);
memcpy(dnsHeaderBuffer, dnsQuery, 12);
dnsHeader = (struct dnsHeader*)dnsHeaderBuffer;

//ignore everything that isn't a DNS query.top bit flag will be 0x0, for 'a name service query'
if(0 != ((ntohs(dnsHeader->flags)) & (1<<15))){
  if(dnsHeaderBuffer != NULL)
    _FREE(dnsHeaderBuffer, 0);
  lockdown(glock);
  return 0;
}

//ignore any packets that don't have queries.
if(0 == ntohs(dnsHeader->qdcount)){
  if(dnsHeaderBuffer != NULL)
    _FREE(dnsHeaderBuffer, 0);
  lockdown(glock);
  return 0;
}

//203 characters is the maximum length of full domain name
void queries = _MALLOC(ntohs(dnsHeader->qdcount)*206, 0, M_ZERO | M_WAITOK);
if(queries != NULL){
  if(dnsHeaderBuffer != NULL)
    _FREE(dnsHeaderBuffer, 0);
  lockdown(glock);
  return 0;
}
```

The variable memBuf is the data buffer of DNS query. We use the function mbuf_copydata to copy the data the buffer receives to the local data buffer allocated by us. I defined a structure to represent the DNS header, and its definition is shown below.

```c
struct dnsHeader {
  u_int16_t id;
  u_int16_t id;
  u_int16_t flags;
  u_int16_t qflags;
  u_int16_t qcount;
  u_int16_t ancount;
  u_int16_t nscount;
  u_int16_t arcount;
};

unsigned char *offset = dnsQuery + @x8c;
unsigned int slen = 0;
unsigned int qIndex = 0;

for(int i=0; i < ntohs(dnsHeader->qdcount); i++){
  slen = (u_int8_t)(offset);
  while(slen != 0){
    memcpy(queries+qIndex, offset+1, slen);
    qIndex += qIndex + slen;
    offset += slen + 1;
  }
}
```

```c
*(char *)queries+qIndex)=0x20;
qIndex += qIndex + 1;
offset += offset + 5;
```
Implemetation to FortiAppMonitor (Monitor network activities using Socket Filters)

• Intercept incoming data using callback `monitorAgent_sflt_data_in`. In this callback, I’m only interested in intercepting the data packets of DNS response. Our goal is to get IP:URL mappings from DNS response.

```c
//ignore everything that isn't a DNS response, top bit flag will be 0x1, for a name service
if(0 == ((ntohs(dnsHeader->flags)) & (1<<15)))
{
    if(dnsHeaderBuf != NULL)
    {
        _FREE(dnsHeaderBuf, 0);
    }
    lck_mtx_unlock(glock);
    return 0;
}

//ignore any packets that don't have answers
if(0 == ntohl(dnsHeader->ancount))
{
    if(dnsHeaderBuf != NULL)
    {
        _FREE(dnsHeaderBuf, 0);
    }
    lck_mtx_unlock(glock);
    return 0;
}

//ignore everything that isn't a DNS response, top bit flag will be 0x1, for a name service
void *queries = _MALLOC(ntohs(dnsHeader->queries) & (1<<15));
if(queries == NULL)
{
    if(dnsHeaderBuf != NULL)
    {
        _FREE(dnsHeaderBuf, 0);
    }
    lck_mtx_unlock(glock);
    return 0;
}

unsigned char *offset = dnsResponse + u_int32_t len = 0;
uint qIndex = 0;

for(int i=0; i<ntohs(dnsHeader->qcount); ++i)
{
    len = u_int16_t(offset) + 2;
    while(len != 0)
    {
        memcpy(queries + qIndex, offset, 2);
        qIndex += 2; len += 2;
    }
}

//ignore everything that isn't a DNS response, top bit flag will be 0x1, for a name service
unsigned char *offset = dnsResponse + u_int32_t len = qIndex + 1;
while(len != 0)
{
    memcpy(queries + qIndex, offset, len);
    qIndex += len; len = 0;
}
}
```

we check to see if the top bit of the member variable `flags` of structure `dnsHeader` is equal to 0x1. If so, it represents that the data packet is a name service response for DNS. We then check the count of answers in the DNS header.
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• For other UDP packets not on port 53, we currently just record the remote IP address and port number.

• Can also monitor the ICMP data packets in the callback monitorAgent_sflt_data_out.

• We discussed the key technical details regarding how to monitor network activities using Socket Filters. This is a powerful module for monitoring the malicious network behaviors of malware on macOS, and it’s easy to recognize the C2 server for malware using this tool.
Implementation to FortiAppMonitor
(Monitor network activities using Socket Filters)

• Demo of monitoring network activities
Implemetation to FortiAppMonitor
(Monitor process exit using KQueue)

• So far, I have discussed how to monitor process execution, file system events, dylib loading event, KEXT loading and unloading events using MACF and monitor network activities using Socket Filters. Both are the kernel APIs, it means that the KEXT module developed using them resides in *kernel space*.

• There are some different ways to monitor process exit, it refers to “Observing Process Lifetimes Without Polling” from Apple.

• Both NSWorkspace and Carbon Event Manager provide a very easy way for you to learn about applications being launched and quit. But both of them only work within a single GUI login context. If you're writing a program that does not run within a GUI login context (a daemon perhaps), or you need to monitor a process in a different context from the one in which you're running, you will need to consider alternatives.
Implemetation to FortiAppMonitor
(Monitor process exit using KQueue)

• One such alternative is the kqueue NOTE_EXIT event. You can use this to detect when a process quits, regardless of what context it's running in. Unlike NSWorkspace and Carbon events, you must specify exactly which process to monitor; there is no way to be notified when any process terminates.

• After comparision, choose kqueue(a kernel event notification mechanism) to monitor process exit. It refers to KQUEUE(2). For monitoring any process terminates using kqueue, you need to do more work on it.
  1. Create a big thread pool, and put the tasks of all running processes in system into the thread pool.
  2. For new process execution, add it into the thread pool to monitor.
  3. If you worry the size of thread pool is not enough to offer all running processes, you can use multiple threading program to do it through launching a new thread to serve on a every single running process in system.
The **kqueue()** system call provides a generic method of notifying the user when an event happens or a condition holds, based on the results of small pieces of kernel code termed filters. A kevent is identified by the (ident, filter) pair. The **kevent()** system call is used to register events with the queue, and return any pending events to the user.
Implementation to FortiAppMonitor
(Monitor process exit using KQueue)

• Use kqueue to develop the module in user space not kernel space to monitor process exit. The details of implementation is shown below.

```c
kq = kqueue();
if (kq == -1) {
    err = errno;
    return;
}
printf("WATCHING PID: %d\n", pid);
EV_SET(&event[0], pid, EVFILT_PROC, EV_ADD, NOTE_EXIT, 0, NULL);
EV_SET(&event[1], 0, EVFILT_USER, EV_ADD, 0, 0, (void*)data);
res = kevent(kq, event, 2, NULL, 0, NULL);
if (res == -1) {
    err = errno;
    printf("Error kevent(): %s\n", err, strerror(err));
    if (strstr(strerror(err), "No such process") != NULL) {
        // the process exits very shortly after execution.
        exited = 1;
        return;
    }
}
while (!closed()) {
    kevent(kq, NULL, 0, events, 2, NULL);
    if (events[0].flags & NOTE_EXIT) {
        exited = 1;
        return;
    }
    if (events[1].filter == EVFILT_USER && events[1].ident == 0 && *((int*)events[1].u.data) == data){
        exited = 0;
        break;
    }
}
return;
```

The call kevent() is blocked until at least one registered event with the queue is received.
Implemetation to FortiAppMonitor
(Monitor process exit using KQueue)

• As I mentioned before, we monitored process execution using MACF. Once a new process is executed, we add its pid into a thread pool to monitor its exit using kqueue.

• For all running processes in system, this utility doesn’t support to monitor all running processes, and only monitor the process exit for new execution. In the future version, it will support to monitor both all existing running processes in system and new processes being executed.
Implementation to FortiAppMonitor
(Monitor process exit using KQueue)

• Demo of monitoring process exit
Implementation to FortiAppMonitor
(Communication between kernel space and user space)

• Beside monitoring process exit using kqueue, other monitoring modules reside in kernel space in form of a kernel extension (KEXT), we also need to send the info collected from KEXT to user space for showing them to users.

• It involves the communication between kernel space and user space. There are several ways to communicate between kernel-mode and user-mode, such as Mach messages, RPC-Mig, BSD syscalls, and memory mapping, IOUserClient, sockets, and the Kernel Control API, etc.

• I chose the kernel control API, which is a socket-based API that allows you to communicate with and receive broadcast notifications from the KEXT. The kernel control (kernel_control) API, which uses the SYSPROTO_CONTROL protocol, allows applications to configure and control a KEXT. And it’s also a bidirectional communication mechanism between a user space application and a KEXT.
Implementation to FortiAppMonitor
(Communication between kernel space and user space)

• For detailed usage of kernel_control API, please see: https://developer.apple.com/library/content/documentation/Darwin/Conceptual/NKEConceptual/control/control.html.

• The kernel process can call a number of functions to send data back to the user space process. This data can be read from the user process using read or recv system calls. In particular, you can use `ctl_enqueue_data` to queue up data to send to the user space process, and `ctl_getenqueue_space` to find out how much free space is available in the queue.
Implementation to FortiAppMonitor
(Communication between kernel space and user space)

**Kernel Space**

ctl_register(&gKernCtlReg, &gKernCtlRef);

ctl_getenqueue(gConn, gUnit, &queueRoom)

queueRoom > msgBufLen

ctl_enqueue(gConn, gUnit, msgBuf, msgBufLen, CTL_DATA_EOR);

**User Space**

socket(PF_SYSTEM, SOCK_DGRAM, SYSPROTO_CONTROL)

ioctl(gSocket, CTLIOCGINFO, &ctl_info)

struct sockaddr_ctl sc;

// Initialize the variable sc
sc.sc_len = sizeof(struct sockaddr_ctl);
sc.sc_family = AF_SYSTEM;
sc.ss_sysaddr = AF_SYS_CONTROL;
sc.sc_id = ctl_info.ctl_id;
sc.sc_unit = CTL_UNIT;

connect(gSocket, (struct sockaddr *)&sc, sizeof(sc))

while(!closed())
{
    memset(recvBuf, 0, sizeof(recvBuf));
    recvLen = recv(gSocket, recvBuf, sizeof(recvBuf), 0);

    msg_kern = (struct k3vinlusec_msg_kern *)recvBuf;

    // Do something to handle the formatted message from kernel

    Send msg
}

Send msg to Kernel Space

User Space
We can set the buffer size for sending data using `ctl_sendsize`. In my KEXT, I set the send size as 4096K to ensure there is enough space when a large number of monitoring events are triggered. You can set any reasonable value on `ctl_sendsize` depending on your needs.
Implementation to FortiAppMonitor
(Communication between kernel space and user space)

- What data is packed into the message? It includes timestamp, pid, ppid, uid, process name, parent process name, buffer and extended buffer depending on event type.

```c
extern void clock_get_calendar_nanotime(
    clock_sec_t *secs,
    clock_nsec_t *nanosecs);

//get uid
uid = kauth_getuid();

//get pid
pid = proc_selfpid();

//get ppid
ppid = proc_selfppid();

//get process name
proc_selfname(procName, 256);

//get parent process name
proc_name(ppid, pprocName, 256);
```
Implementation to FortiAppMonitor
(Communication between kernel space and user space)

• Note: Some events can be triggered in a very short time (the same event on the same vnode triggers ~100k times per second), such as file open, read and write operations on some specific vnodes (/dev/oslog_stream, /dev/ptmx, /dev/urandom, /dev/.* etc.). It could fully queue the send buffer shortly. It causes the message for other triggered events cannot be sent to user space.

• You need to filter the large of triggered events using two ways.
  1. Filter out all file open, read and write events on these specific vnodes and DON’T send the message to user space.
  2. Maintain a global table to record the timestamp for file open, read and write operation in order to filter the triggered event in a very short time. I choose this way!
Implemetation to FortiAppMonitor (Communication between kernel space and user space)

• Filtering Algorithm

```c
double timestamp = secs + (1.0*nanosecs)/1000000000;
bool isSend = searchTable(timestamp, procName, path, FILE_OPEN_EVENT);
if(isSend)
{
    if(msgBuf != NULL)
    {
        _FREE(msgBuf, 0);
    }
    lck_mtx_unlock(gLock);
    return 0;
}
```

```c
struct statustTable{
    double timestamp;
    char processname[256];
    char path[1024];
    uint8_t etype;
};
```

```c
#define STABLELEN 512
struct statustTable gStatusTable[STABLELEN];
```

```c
for(int i=0; i<(gTableCount)STABLELEN ? STABLELEN : gTableCount; i++){
    if(strcmp(gStatusTable[i].processname, procname)==0 & & strcmp(gStatusTable[i].path, path)==0 & & gStatusTable[i].etype == etype){
        timediff = timestamp - gStatusTable[i].timestamp;
        //printf("timediff: %f\n", timediff);
        if(timediff < 0.01){
            //update timestamp, and return false to not send the record to GUI.
            gStatusTable[i].timestamp = timestamp;
            return false;
        }else{
            gStatusTable[i].timestamp = timestamp;
            return true;
        }
    }
}
```

You can set the time difference depending on your circumstance, here I set it as 0.01s
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowledgement
Case study

• OSX.Dummy – A new mac malware targets the cryptocurrency community. The detailed info refers to https://objective-see.com/blog/blog_0x32.html and https://isc.sans.edu/diary/23816

• Analyze the behaviors of OSX.Dummy malware using our utility.

• Launch the malware using “chmod +x script && ./script”, the file script is mach-o format.
Case study

- Write file to “/private/tmp/script.sh”, /tmp is a symbolic link to /private/tmp, so also write file to “/tmp/script.sh”
Case study

- Write file to “/private/tmp/dumpdummy” and “/Users/Shared/dumpdummy”
Case study

- Change the owner of script.sh to root, and then add executable privileges for it.
Case study

- Move file /tmp/script.sh to /var/root/script.sh
Case study

- Write file /private/tmp/com.startup.plist, and move it to /Library/LaunchDaemons/
Case study

- File move and rename operations

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Process Execution</th>
<th>mv</th>
<th>9918</th>
<th>root</th>
<th>Process E.</th>
<th>&quot;parent process&quot;: &quot;sudo&quot;, &quot;ppid&quot;: 9916, &quot;uid&quot;: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1089</td>
<td>1532713919.271076807</td>
<td>mv</td>
<td>9918</td>
<td>root</td>
<td>Process E.</td>
<td>&quot;parent process&quot;: &quot;sudo&quot;, &quot;ppid&quot;: 9916, &quot;uid&quot;: 0</td>
</tr>
<tr>
<td>1090</td>
<td>1532713919.272050810</td>
<td>File Open</td>
<td>mv</td>
<td>9918</td>
<td>root</td>
<td>Process E.</td>
</tr>
<tr>
<td>1091</td>
<td>1532713919.274254000</td>
<td>Process Exit</td>
<td>mv</td>
<td>9918</td>
<td>root</td>
<td>Process E.</td>
</tr>
<tr>
<td>1092</td>
<td>1532713919.273844253</td>
<td>File Rename</td>
<td>mv</td>
<td>9918</td>
<td>root</td>
<td>Process E.</td>
</tr>
<tr>
<td>1093</td>
<td>1532713919.282157000</td>
<td>Process Exit</td>
<td>sudo</td>
<td>9916</td>
<td>N/A</td>
<td>Process E.</td>
</tr>
<tr>
<td>1094</td>
<td>1532713919.285861000</td>
<td>Process Exit</td>
<td>ps</td>
<td>9914</td>
<td>N/A</td>
<td>Process E.</td>
</tr>
<tr>
<td>1095</td>
<td>1532713919.285885000</td>
<td>Process Exit</td>
<td>sh</td>
<td>9914</td>
<td>N/A</td>
<td>Process E.</td>
</tr>
</tbody>
</table>

```

```

<table>
<thead>
<tr>
<th>Record ID</th>
<th>File Rename</th>
<th>mv</th>
<th>9918</th>
<th>root</th>
<th>Rename f...</th>
<th>&quot;parent process&quot;: &quot;sudo&quot;, &quot;ppid&quot;: 9916, &quot;uid&quot;: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1092</td>
<td>1532713919.273844253</td>
<td>File Rename</td>
<td>mv</td>
<td>9918</td>
<td></td>
<td>&quot;parent process&quot;: &quot;script&quot;, &quot;ppid&quot;: 9874, &quot;uid&quot;: 20</td>
</tr>
<tr>
<td>1093</td>
<td>1532713919.282157000</td>
<td>Process Exit</td>
<td>sudo</td>
<td>9916</td>
<td>N/A</td>
<td>&quot;parent process&quot;: &quot;script&quot;, &quot;ppid&quot;: 9874, &quot;uid&quot;: 20</td>
</tr>
<tr>
<td>1094</td>
<td>1532713919.285861000</td>
<td>Process Exit</td>
<td>ps</td>
<td>9914</td>
<td>N/A</td>
<td>&quot;parent process&quot;: &quot;script&quot;, &quot;ppid&quot;: 9874, &quot;uid&quot;: 20</td>
</tr>
<tr>
<td>1095</td>
<td>1532713919.285885000</td>
<td>Process Exit</td>
<td>sh</td>
<td>9914</td>
<td>N/A</td>
<td>&quot;parent process&quot;: &quot;script&quot;, &quot;ppid&quot;: 9874, &quot;uid&quot;: 20</td>
</tr>
</tbody>
</table>
Case study

- Change the owner of `/Library/LaunchDaemons/com.startup.plist` to `root`
Case study

- Load com.startup.plist as a daemon service
Case study

- Execute the python script, and connect to the C2 server 185.243.115.230:1337
Case study

• At this point, we finished analyzing the behaviors of malware OSX.Dummy using the utility
• It’s easy to quickly identify the behaviors of malware OSX.Dummy in this case study.
• It’s more helpful for security researchers or malware analysts to analyze the malware samples on macOS.
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowleage
Conclusion

• The Mandatory Access Control Framework offers the more detailed granularity than KAuth. It’s perfectly suited for third party security products. It’s a good choice to implement monitoring kinds of events.

• Monitor network activities using Socket Filters, you can do some work on filtering more fine-grained protocols based on UDP and TCP.

• The kevent and kqueue APIs provide a simple and efficient mechanism for event notification. It’s used to monitor the process exit.

• Friendly UI design developed using Electron and W2UI, it provides a powerful search and filter functionality, as well as fine-grained monitoring events. You can also save the records into a .json format log file. Support to copy one specific record using shortcut key “Command+C” to clipboard in GUI.
Conclusion

• Will be released as a freeware in the coming week after Black Hat USA 2018 Arsenal.
• You’re very welcome to send the feedbacks, suggestion or bugs to kailu@fortinet.com or Twitter: @k3vinlusec.
• Discuss all details of implementation, you can build and develop your own arsenal based on the technical details.
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowleage
References

- https://objective-see.com/blog.html
- MacOS and iOS Internals, Volume III: Security & Insecurity By Jonathan Levin
- Mac OS X Internals: A Systems Approach By Amit Singh
- OS X and iOS Kernel Programming by Ole Henry Halvorsen | Douglas Clarke
- https://github.com/objective-see/LuLu/
- https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2
- https://wiki.netbsd.org/tutorials/kqueue_tutorial/
- http://w2ui.com/web/docs/1.5/
- https://electronjs.org/docs
Agenda

• BIO
• Introduction to FortiAppMonitor
• Implementation to FortiAppMonitor
• Case study
• Conclusion
• Reference
• Acknowledgement
Acknowleage

• Thanks to Weining Wu, Peixue Li, Zhouyuan Yang, Linan Liu.
Thank You!

Kai Lu (@k3vinlusec)
Email: kailu@fortinet.com