Simple HIDS(Host Intrusion Detection System)

본 문서는 Simple Host Intrusion Detection System 이하 (Simple HIDS)를 설명한 문서입니다.

> 공식 제공 문서가 아니라, 소스코드를 보면서 분석한 문서이기에 원래 코드와는 조금 다른 해석이 있을 수 있습니다.

Index

본 문서의 목차는 다음과 같습니다:

> 하이퍼링크의 경우 뷰어 환경에 따라 제대로 동작하지 않을 수 있습니다.

Background

Library

> `<stdio.h>` , `<stdlib.h>` 및 흔하게 많이 사용되는 C 라이브러리는 명시하지 않았습니다.

- **inotify**: `inotify` 를 통해 파일 이벤트를 모니터링합니다. 이는 소스코드 내부에서 `<sys/inotify.h>` 를 `include` 하여 사용하고 있습니다.
- **openssl**: `openssl` 을 통해 MD5와 SHA Checksum을 계산합니다. 이는 이후 파일이 변조되었는지 아닌지에 대해서 확인할 때 사용합니다.
- **thpool**: `thpool.h` 와 `thpool.c` 를 통해 thread pool을 사용합니다.
- **pthread**: 다중 디렉토리 감시 (원래 소스코드에서는 포함되지 않았지만) 및 thread pool을 구현하기 위해 `<pthread.h>` 를 `include` 하여 사용하고 있습니다.

Design

Simple HIDS는 다음과 같이 동작합니다.

1. `inotify` 를 통해 호스트의 시스템에서 파일 이벤트를 모니터링 합니다.
2. 모니터링된 파일 이벤트를 바탕으로 기존의 파일이 수정되었는지를 checksum을 통해 확인하거나, 새로운 파일이 추가되었다면 모니터링할 파일을 추가하는 등의 적절한 조치를 취합니다.
3. 파일 이벤트에 대한 내용을 유저에게 stdout으로 출력합니다.

Makefile

Usage

Simple HIDS는 간단한 Makefile을 제공하며, 다음의 receipe를 사용할 수 있습니다:

- `clean` : 빌드 된 실행 파일과, 컴파일된 `.o` 파일들이 있는 디렉토리를 삭제합니다. 예시 명령어는 다음과 같습니다.

```
make clean
```

- `all` : `simple_nids` 를 빌드합니다. 예시 명령어는 다음과 같습니다.

```
make all
```

Code

> 일부 주요 코드만 분석하였습니다, `all: $(PROG)` 와 같은 단순 정의의 경우 설명을 생략했습니다. `Makefile` 의 코드에 대한 분석입니다.

```
.PHONY: all clean
```

`all` 과 `clean` 레시피를 재정의합니다.

```
CC = gcc
```

`gcc` 를 컴파일러로 사용합니다.

```
CFLAGS = -O2 -Wall -std=gnu99
```

```
LDFLAGS = -lpthread -lssl -lcrypto
```

`gcc` 를 이용해 컴파일 및 링킹 작업을 할 시, 사용할 flag들을 설정합니다. 링킹 과정에서 멀티쓰레드를 위해 `-lpthread` 옵션을, checksum 계산을 위해 `-ssl` 그리고 `-lcrypto` 옵션을 사용합니다,

```
SRC = $(shell find . -name '*.c')
OBJ = $(patsubst %.c, %.o, $(SRC))
```

소스코드 디렉토리에 존재하는 모든 `.c` 파일들을 찾고 저장합니다. 또한 컴파일 후 저장할 object 파일들의 이름들을 설정합니다. 예를 들어, `decoder.c` 는 `decoder.o` 로 컴파일 할 수 있게 설정합니다. Simple NIDS랑은 다르게, Simple HIDS는 단일 디렉토리에 모든 소스코드가 존재합니다.

```
$(PROG): $(OBJ)
        $(CC) -o $@ $^ $(LDFLAGS)
```

Target 인 `$(PROG)` 를 빌드할 시 사용할 prerequsite들에 대한 정의를 진행합니다. 해당 Target의 경우, 컴파일 된 모든 `.o` 파일들을 링킹하는 과정을 가지고 있습니다. `$(CC) -o $@ $^ $(LDFLAGS)` 를 통해, 모든 `.o` 파일들을 `simple_hids` 실행 파일로 컴파일 합니다.

```
%.o: %.c
        $(CC) $(CFLAGS) -o $@ -c $<
```

Target인 `%.o` 를 빌드할 시 사용할 prerequsite들에 대한 정의를 진행합니다. 예를 들어, 만일 `decoder.o` 라는 target이 존재할 시, 다음의 명령어를 통해 컴파일 합니다:

```
gcc -O2 -Wall -std=gnu99 -o decoder.o -c decoder.c
```

이와 같이, 사전에 정의된 모든 `$(OBJ)` 의 object 파일에 대해 컴파일을 진행합니다. 컴파일 된 모든 object 파일들은, 이후 `simple_hids` 를 컴파일 하며 사용하게 됩니다.

```
clean:
        rm -rf $(PROG) $(OBJ)
```

Receipe `clean` 을 추가합니다. 이는 `simple_hids` 실행 파일과, 모든 object 파일이 들어있는 디렉토리를 삭제합니다.

Main

Simple HIDS 소스코드 중, `main.c` 의 소스코드에 대한 설명입니다.

main.c

`main/main.c` 에는 다음의 함수들이 구현되어있습니다:

`void sigint_handler(int signo)`

`SIGINT` 가 발생했을 때, 프로그램을 정상 종료하기 위한 signal handler 입니다. 이는 `RUN` flag를 `0` 으로 설정하여 inotify가 loop를 그만하도록 만듭니다.

`int main(int argc, char **argv)`:

`simple_hids` 프로그램이 시작할 때 호출되는 함수입니다. 이는 다음의 함수들을 순차적으로 호출합니다. 각 함수에 대한 설명은 이후 다시 자세하게 설명됩니다.

1. `load_entries()` : 대상 디렉토리에 있는 모든 파일들에 대한 `struct entry` 를 로드합니다 .
2. `signal : sigint_handler` 를 `SIGINT` 의 signal handler로 설정합니다.
3. `pthread_create` 디렉토리를 watch하는 thread를 생성합니다. 이후 `RUN` flag 가 `0` 이 될때까지 loop를 반복합니다.

Common

Simple HIDS 소스코드 중, `common.h` 의 소스코드에 대한 설명입니다.

common.h

`common.h` 는 다음의 기능을 합니다.

- 필요한 헤더 파일들 include
- 특정 값들 define
- `struct entry` 정의
- 함수들에 대한 선언

`struct entry`

Simple HIDS에서는 각 파일들을 `struct entry` 로 관리합니다. `struct entry` 의 경우, left chid right sibling tree를 이용하여 표현합니다.

```
struct entry {
        char name[MAX_STRING]; // name
        mode_t mode; // mode
        off_t size; // total size
        time_t atime; // time of last access
        time_t mtime; // time of last modification
        unsigned char hash[MD5_DIGEST_LENGTH];
        struct entry *sibling;
        struct entry *child;
};
```

`struct entry` 는 위와 같이 선언되어있습니다. `struct entry` 의 경우, linked list 형식으로 sibling과 child를 저장합니다.

`struct thread_arg`

Thread pool에 추가할 작업의 argument를 저장하기 위해서 `struct thread_arg` 를 사용합니다.

```
struct thread_arg {
        int *RUN;
        char *root;
        struct entry *entries;
};
```

`struct thread_arg` 은 위와 같이 선언되어있습니다.

Hash

Simple HIDS 소스코드 중, `hash.c` 그리고 `hash.h` 에 대한 설명입니다. `hash.h` 의 경우, 단순한 `#include` 이므로 설명을 생략합니다.

hash.c

`hash.c` 은 다음의 함수들을 구현합니다.

`int hash_func(char *path, unsigned char *hash)`

특정 파일의 MD5 hash를 계산하고 반환하는 함수입니다.

```
MD5_CTX ctx;
MD5_Init(&ctx);
```

위의 코드를 통해 MD5를 초기화 합니다.

```
int bytes;
unsigned char data[1024];
while ((bytes = fread(data, 1, 1024, fp)) != 0)
        MD5_Update(&ctx, data, bytes);
```

이후 파일에 대한 MD5 checksum을 저장합니다.

```
MD5_Final(hash, &ctx);
fclose (fp);
```

MD5 checksum을 계산한 이후, 파일을 닫습니다. 이렇게 계산된 MD5 checksum은 이후 `inotify` 의 파일 이벤트 탐지 이후 사용하게 됩니다.

Entry

Simple HIDS 소스코드 중, `entry.c` 그리고 `entry.h` 에 대한 설명입니다. `entry.h` 의 경우, 단순한 `#include` 이므로 설명을 생략합니다.

Background

Simple HIDS에서는 각 파일 하나 하나를 `struct entry` 로 저장합니다. 이는 `common.h` 에 정의되어있습니다.

entry.c

`entry.c` 은 다음의 함수들을 구현합니다.

`int update_entry_info(struct entry *node, const char *path)`

한 `struct entry` 에 필요한 정보를 저장합니다. 필요한 정보를 저장하기 위해 다음의 절차를 수행합니다.

1. `struct stat` 을 저장하여 파일의 정보를 저장합니다.

```
if (stat(path, &info) != 0) {
        printf("Failed to get the stat of %s\n", path);
        return -1;
}
```

2. `stat` 으로 저장된 정보를 `struct entry` 에 맞게 저장합니다.

```
node->mode = info.st_mode;
node->size = info.st_size;
node->atime = info.st_atime;
node->mtime = info.st_mtime;
```

3. 이후 `hash_func` 을 통해 hash 값을 계산하고, 이를 저장합니다.

```
hash_func(node->name, node->hash);
```

4. Child와 sibling을 초기화합니다.

```
node->sibling = NULL;
node->child = NULL;
```

이 함수를 통해 하나의 파일에 대해 하나의 `struct entry` 정보를 저장할 수 있게 됩니다.

`struct entry * update_entries(struct entry *head, const char *current_dir)`

현재 디렉토리로부터 재귀적으로 모든 파일들을 하나씩 load하고, `struct entry` 의 형태로 각 파일들을 저장합니다. 이 함수는 다음과 같이 동작합니다.

1. 현 디렉토리의 모든 파일들에 대해서 반복하여 실행합니다.

```
struct dirent *entry;
while ((entry = readdir(dp)) != NULL) {
```

2. 각 파일에 대해서 `struct entry` 를 하나씩 생성하고, entry 정보를 저장합니다.

```
struct entry *new_node = (struct entry *)malloc(sizeof(struct entry));
if (new_node == NULL) {
        printf("Failed to allocate a new node\n");
        continue;
}
update_entry_info(new_node, path);
```

3. 이후 추가적인 정보(access time과 modified time)를 저장합니다.

```
char atime[MAX_STRING];
sprintf(atime, "%s", ctime(&new_node->atime));
atime[strlen(atime)-1] = 0;

char mtime[MAX_STRING];
sprintf(mtime, "%s", ctime(&new_node->mtime));
mtime[strlen(mtime)-1] = 0;
```

4. 만일 현재 파일이 디렉토리이면, `new_node->child` 에 child 를 추가합니다. Recursive 하게 child 파일을 저장할 수 있도록 호출합니다. 이후 저장된 `struct entry` 에 대한 정보를 출력합니다.

```
if (S_ISDIR(new_node->mode)) {
        printf("d: %s | %lld | %s | %s | %02x\n", path, (long
long)new_node->size, atime, mtime, new_node->hash[0]);
        new_node->child = update_entries(new_node->child, path);
}
```

5. 만일 현재 파일이 디렉토리가 아니라 일반 파일이였다면, 화면에 `struct entry` 에 대한 정보를 출력만 합니다.
6. `struct entry` 를 전체 파일을 저장하는 left child right sibling tree에 추가합니다.

```
if (head == NULL) {
        head = new_node;
} else {
        new_node->sibling = head;
        head = new_node;
}
```

`struct entry *load_entries(struct entry *head, char *dir)`

특정 디렉토리로부터 모든 subdirectory의 모든 파일들을 저장합니다. 다음의 절차를 수행합니다.

1. `head` 를 `struct entry` 로 초기화

```
head = (struct entry *)malloc(sizeof(struct entry));
```

2. 첫 `head` 를 시작으로 `update_entries` 를 재귀적으로 진행합니다.

```
update_entry_info(head, dir);
```

```
head->child = update_entries(head->child, dir);
```

맨 처음 `head` 만 추가하면, 알아서 subdirectory 등을 recursive 하게 load 하기 때문에, 걱정할 필요가 없습니다.

`int release_entries(struct entry *head)`

저장된 `struct entry` 들을 일괄적으로 삭제합니다.

```
while (peer != NULL) {
        struct entry *temp = peer;
        peer = peer->sibling;
        if (S_ISDIR(temp->mode)) {
                release_entries(temp->child);
                printf("(release) d: %s\n", temp->name);
        } else {
                printf("(release) f: %s\n", temp->name);
        }
        free(temp);
}
```

현재 `struct entry` 의 sibling을 하나씩 하나씩 순회하면서 모든 `struct entry` 를 삭제합니다. Sibling이 디렉토리라면, child를 추가적 재귀 함수를 통해 삭제합니다. 만일 sibling이 단순 파일이라면 `free` 를 통해 삭제합니다.

`int add_entry(struct entry *head, const char *path)`

특정 `struct entry` 에 입력받은 파일에 대한 `struct entry` 를 추가합니다. 이 함수는 다음의 절차를 수행합니다.

1. 현 디렉토리에 있는 모든 sibling에 대해서 순회합니다. 반복을 진행하며 현재 `peer` 과 이전의 `peer` 인 `prev` 를 저장합니다.

```
while (peer != NULL) {
...
        prev = peer;
        peer = peer->sibling;
```

2. 만일 현재 sibling이 디렉토리인 경우, 해당 디렉토리에 들어가는 파일이라면 sibling에 대상 파일을 추가합니다. 당연히 새로 추가되는 파일의 경우 또한 `add_entry` 를 통해 파일을 저장합니다.

```
if (S_ISDIR(peer->mode)) {
        if (strncmp(peer->name, path, strlen(peer->name)) == 0) {
```

```
                printf("lookup: %s\n", peer->name);
                if (add_entry(peer->child, path) == 0) {
                        return 0;
                }
        }
}
```

예를 들어, 현재 sibling이 `/foo/bar` 인 경우, 그리고 새로 입력 받은 파일이 `/foo/bar/baz` 인 경우, `/foo/bar/baz` 는 `/foo/bar` 의 child로 저장이 됩니다. 3. 만일 현재 sibling과 추가되는 파일의 이름이 동일할 경우, add가 아닌 update를 실행합니다.

```
else { // file
        if (strcmp(peer->name, path) == 0) {
                printf("update (not add): %s\n", peer->name);
                update_entry_info(peer, path);
                return 0;
        }
}
```

예를 들어, 현재 sibling이 `/foo/bar` 인 경우, 그리고 새로 입력받은 파일 또한 `/foo/bar` 인 경우 `/foo/bar` 을 add 하지 않고, 단순히 update 만 진행합니다.

4. 만일 loop가 진행되면서도 return을 하지 못한 경우, 새로운 `struct entry` 를 생성하고 정보를 저장합니다.

```
struct entry *new_node = (struct entry *)malloc(sizeof(struct entry));
if (new_node == NULL) {
        printf("Failed to allocate a new node\n");
        return -1;
}
update_entry_info(new_node, path);
```

5. 이후 `prev` 의 sibling으로 새로 탐색된 파일을 저장합니다.

```
if (head == NULL) {
        head = new_node;
} else {
        prev->sibling = new_node;
}
```

`int update_entry(struct entry *head, const char *path)`

기존에 존재하는 `struct entry` 를 update하는 함수입니다.

1. 현재 `head` 에 대해서 모든 sibling을 순회합니다.

```
while (peer != NULL) {
...
peer = peer->sibling;
```

2. 만일 현재 sibling이 디렉토리인 경우, 업데이트 하고자 하는 파일이 현재 sibling 디렉토리에 존재하는 파일인지 확인합니다. 만일 현재 sibling 디렉토리에 존재하는 파일이라면, 재귀함수를 호출합니다.

```
if (S_ISDIR(peer->mode)) {
        if (strncmp(peer->name, path, strlen(peer->name)) == 0) {
                printf("lookup: %s\n", peer->name);
                update_entry(peer->child, path);
        }
}
```

예를 들어, 업데이트 하고자 하는 파일이 `/foo/bar/baz` 이면, 처음에는 `/foo` 디렉토리, 이후는 `/foo/bar` 디렉토리를 재귀함수로 탐색합니다. 3. 만일 현재 sibling이 일반 파일인 경우, 업데이트하고자 하는 파일과 이름이 같은지 확인하고 만일 이름이 같다면 `update_entry_info` 를 통해 업데이트를 진행합니다.

```
if (strcmp(peer->name, path) == 0) {
        printf("update: %s\n", peer->name);
        update_entry_info(peer, path);
        return 0;
}
```

`int remove_entry(struct entry *head, const char *path)`

특정 `struct entry` 를 삭제하는 함수입니다. 대상 파일에 해당하는 `struct entry` 를 삭제하기 위해 다음의 절차를 수행합니다.

1. `head` 의 모든 sibling을 순회하며 반복합니다. 또한 마찬가지로 `prev` 와 `peer` 를 저장합니다.

```
while (peer != NULL) {
...
        prev = peer;
        peer = peer->sibling;
}
```

2. 만일 현재 sibling이 디렉토리인 경우, 해당 디렉토리의 파일이 맞는지를 확인합니다. 만일 현재 sibling에 대상 파일이 존재하는 경우, 재귀함수를 호출합니다.

```
if (S_ISDIR(peer->mode)) {
        if (strncmp(peer->name, path, strlen(peer->name)) == 0) {
                printf("lookup: %s\n", peer->name);
                remove_entry(peer->child, path);
        }
}
```

예를 들어, 삭제하고자 하는 파일이 `/foo/bar/baz` 이면, 처음에는 `/foo` 디렉토리, 이후는 `/foo/bar` 디렉토리를 재귀함수로 탐색합니다.

3. 만일 현재 sibling이 파일인 경우, 삭제하고자 하는 파일인지 확인합니다. 만일 맞다면 현재 sibling의 `struct entry` 를 제거하고 이전 entry와 이후 entry를 직접 연결합니다.

```
if (strcmp(peer->name, path) == 0) {
        printf("remove: %s\n", peer->name);
        if (prev != NULL) {
                struct entry *temp = peer;
                prev->sibling = peer->sibling;
                free(temp);
        } else {
                head = peer->sibling;
                free(peer);
        }
        return 0;
}
```

예를 들어, `/foo` 의 sibling이 `/bar` 이고, `/bar` 의 sibling이 `/baz` 였고, `/bar` 를 삭제할 시 `/foo` 의 sibling이 직접 `/baz` 로 변합니다. 이를 통해 특정 `struct entry` 를 삭제할 수 있습니다.

Thpool

Simple HIDS 소스코드 중, `thpool.c` 그리고 `thpool.h` 에 대한 설명입니다. 이는 외부 라이브러리를 사용했고, 간단하게 thread pool 을 사용할 수 있게 합니다.

Background

각 파일의 hash를 계산하는 경우, 다수의 파일을 빠르게 효율적으로 처리하기 위해 thread pool을 활용할 수 있습니다.

Watch

Simple HIDS 소스코드 중, `watch.c` 그리고 `watch.h` 에 대한 설명입니다. 이는 `inotify` 를 사용해서 파일 이벤트를 감시합니다.

`static void __handle_inotify_event(struct thread_arg *t_arg, const struct inotify_event *event)`

파일 이벤트가 탐지되었을 시 사용되는 callback 함수입니다. 다음의 이벤트에 대한 처리를 할 수 있습니다:

- `IN_ACCESS`
- `IN_ATTRIB`
- `IN_CREATE`
- `IN_MODIFY`
- `IN_DELETE`
- `IN_DELETE_SELF`
- `IN_MOVED_FROM`
- `IN_MOVED_TO`
- `IN_MOVE_SELF`

`int read_with_timeout(int fd, char *buf, int buf_size, int timeout_ms)`

Timeout 을 가지고 이벤트를 읽어옵니다. 이렇게 사용하는 이유는 기존 `read` 의 경우 blocking을 합니다. 즉 맨 마지막에 프로그램을 종료하기 위해 `RUN` flag를 `0` 으로 설정하는 경우, 자동으로 종료하지 못합니다. 따라서 timeout을 통해서 일부 시간이 지나면 자동으로 read를 종료하는 기능을 추가합니다.

`void *watch_dir(void *arg)`

특정 디렉토리를 `inotify` 를 통해 watch하는 함수입니다.

```
int fd = inotify_init();
...
int wd = inotify_add_watch(fd, t_arg->root, IN_CREATE | IN_MODIFY |
IN_DELETE | IN_MOVED_FROM | IN_MOVED_TO | IN_MOVE_SELF);
```

를 통해 `inotify` 에 대상 디렉토리에 대한 watch를 추가합니다. 이후 `RUN` flag 가 활성화된 경우 지속적으로 `read_with_timeout` 를 호출하여 파일 이벤트를 모니터링 합니다. 또한 이벤트가 발생한 경우, `__handle_inotify_event` 를 통해 callback 함수를 호출합니다.

```
...
int ret = inotify_rm_watch(fd, wd);
...
```

를 통해 더이상 파일을 감시하지 않는 경우 `inotify` 의 watch를 제거합니다.

Limits

Simple HIDS의 한계점을 서술합니다.

다중 디렉토리 동시 모니터링 불가

현재 코드의 경우, 다중 디렉토리를 동시에 모니터링할 수 없습니다. 다중 디렉토리를 동시에 모니터링 하기 위해서는 `watch_dir` 를 multithread로 동작하게 하면 모니터링할 수 있습니다.

새로운 디렉토리 모니터링 불가

현재 코드의 경우, 새로운 subdirectory가 생성된 경우 해당 디렉토리의 파일들은 자동으로 모니터링할 수 없습니다. 이를 해결하기 위해 디렉토리가 생성될 시 `inotify_add_watch` 를 별도로 해주는 것이 필요합니다.