Simple NIDS(Network Intrusion Detection System)

본 문서는 Simple Network Intrusion Detection System 이하 (Simple NIDS)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.

- **pcap**: `libcap` 을 통하여 네트워크 패킷을 캡쳐합니다. 캡쳐된 네트워크 패킷에 대한 추가적인 분석을 진행하여, 네트워크 탐지 시스템을 구현하는데 중요한 기능을 하고 있습니다. 이는 소스코드 내에서 `<pcap.h>` 를 `include` 하여 사용하고 있습니다
- **regex**: RegEx를 통해 탐지된 네트워크 패킷에서 특정 문자열 패턴이 존재하는지 등에 대한 추가적인 분석을 진행합니다. 이는 소스코드 내에서 `<regex.h>` 를 `include` 하여 사용하고 있습니다.

- **pthread**: `lpthread` 를 통하여 packet pool을 구성할 때 사용되는 spin lock을 구현합니다. 이는 소스코드 내에서 `<pthread.h>` 를 `include` 하여 사용하고 있습니다.

Design

Simple NIDS는 다음과 같이 동작합니다.

1. pcap을 통해 저장된 패킷 기록을 재생하거나, 실시간으로 들어오는 패킷을 탐지 (reception)
2. 탐지된 패킷을 전처리 및 디코딩 (decoder)
3. 탐지된 패킷의 내용물을 문자열 매치 또는 RegEx를 통해 분석 (detection)
4. 탐지된 내용이 있다면 파일 또는 stdout 로그로 출력 (log)

Makefile

Usage

Simple NIDS는 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.

```
INC_DIR = include
SRC_DIR = main reception pool decoder rule detection log
OBJ_DIR = obj
```

include 할 디렉토리와, 소스코드를 사용할 디렉토리 및 컴파일 된 object 파일을 저장할 디렉토리를 설정합니다.

```
CFLAGS = -g -Wall -std=gnu99 $(addprefix -I,$(shell find $(SRC_DIR) -type
d)) -DDEBUG
LDFLAGS = -lpthread -lpcap
```

`gcc` 를 이용해 컴파일 및 링킹 작업을 할 시, 사용할 flag들을 설정합니다. 컴파일 옵션 중 `-DDEBUG` 를 추가함으로 소스코드 내부의 `#ifdef DEBUG` 에 해당하는 코드들을 모두 사용합니다. 링킹 과정에서 `pthread.h` 및 `pcap.h` 를 사용하기 위해 각각 `-lpthread` , `-lpcap` 옵션을 설정합니다.

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

```
vpath %.c $(dir $(SRC))
vpath %.o $(OBJ_DIR)
```

`.c` 파일과 `.o` 파일들을 모두 `vpath` 를 이용하여 저장합니다.

```
$(PROG): $(addprefix $(OBJ_DIR)/, $(OBJ))
        mkdir -p $(@D)
        $(CC) -o $@ $^ $(LDFLAGS)
```

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

```
$(OBJ_DIR)/%.o: %.c
        mkdir -p $(@D)
        $(CC) $(CFLAGS) -o $@ -c $<
```

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

```
gcc -g -Wall -std=gnu99 $(addprefix -I,$(shell find $(SRC_DIR) -type d)) -
DDEBUG -o /obj/decoder.o -c /obj/decoder.c
```

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

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

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

Main

Simple NIDS 소스코드 중, `main/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Main의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `main.c` : `SIGINT` 를 위한 signal handler 및 프로그램 초기 실행에 관련된 소스 파일
- `context.c` : 사용자에게 입력받은 command line arguments를 처리하는 소스 파일
- `include/common.h` : 필요한 header 파일들을 include 하고, `#define` 과 `enum` 이 정의된 헤더 파일
- `include/context.h` : 프로그램 전반적으로 사용되는 `struct` 및 함수들이 정의된 헤더 파일

main.c

`main/main.c` 에는 다음의 함수들이 구현되어있습니다:

`void sigint_handler(int signo)`

`SIGINT` 가 발생했을 때, 프로그램을 정상 종료하기 위한 signal handler. 이는 다음의 함수들을 순차적으로 호출합니다:

1. `ctx.reception_destroy_func()` : Simple NIDS의 경우 pcap을 사용하여 패킷을 캡쳐합니다. 따라서 reception을 중지하고 기능들을 멈추기 위해 사용합니다.

2. `destroy_packet_pool()` : packet pool을 통해 캡쳐한 패킷들을 저장 및 처리하기에, 생성된 packet pool을 정상 정지할 수 있게 합니다.
3. `destroy_rule_table()` : rule table에 등록된 모든 rule들을 삭제합니다.

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

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

1. `init_context()` : 프로그램에서 사용되는 `context_t` 를 초기화합니다.
2. `initialize_packet_pool()` : 프로그램에서 사용되는 packet pool을 초기화 합니다.
3. `initialize_rule_table()` : rule table을 초기화합니다.
4. `load_rules()` : `rules.txt` 파일의 rule들을 로딩하여 저장합니다.
5. `signal` : `sigint_handler` 를 `SIGINT` 의 signal handler로 설정합니다.
6. `ctx.reception_func()` : 패킷 캡쳐를 시작합니다.

context.c

`main/context.c` 에는 `init_context()` 가 구현되어있습니다. 이는 유저가 입력한 command line argument를 parsing 하고, `context_t` 에 들어가는 function pointer 등을 유저의 요청에 맞게 설정합니다. 또한, Command line argument를 parsing 하기 위해 `<getopt.h>` 의 `getopt_long` 함수를 사용합니다.

```
static struct option long_options[] = {
        {"interface", required_argument, 0, 'i'},
        {"file", required_argument, 0, 'f'},
        {"match", required_argument, 0, 'm'},
        {"regex", no_argument, 0, 'r'},
        {"log", required_argument, 0, 'l'},
        {0, 0, 0, 0}
};
```

프로그램에서 제공하는 command line arguments는 다음과 같습니다:

- **Interface**: 패킷 캡쳐에서 사용할 interface를 설정합니다. 기본값의 경우 실시간으로 캡쳐되는 패킷을 탐지합니다.
- **File**: 만일 패킷 캡쳐에서 파일을 재생하는 경우, 사용할 `.pcap` 파일을 설정합니다.
- **Match**: String match 에 사용되는 종류를 설정합니다. `string1` 과 `string2` 두가지 종류를 선택할 수 있습니다.
- **RegEx**: String match 대신 RegEx를 사용할지 아닐지에 대한 여부입니다.
- **Log**: 로그를 어떤 방식으로 기록할지에 대한 설정입니다. `stdout` 또는 파일로 기록합니다.

-

context.h

`main/include/context.h` 에는 다음의 내용이 정의되어있습니다.

- `struct packet_t` : 캡쳐된 패킷을 일부 전처리한 구조체. (Linked list 방식으로 동작)
- `struct rule_t` : 캡쳐된 패킷을 탐지할 때 사용할 규칙을 저장하는 구조체. (Linked list 방식으로 동작)
- `struct context_t` : 현재 실행되는 Simple NIDS의 정보 및 설정을 담고 있는 구조체. 또한 사용할 함수들의 function pointer를 저장함. 예를 들어

```
int (*reception_func)(struct context_t *ctx);
```

의 경우, reception할 때 사용되는 함수를 정의합니다. 약간 function pointer를 통해서 polymorphism을 구현하기 위해 사용되는 느낌.

- 이외 사용되는 모든 함수들에 대한 원형 정의.

common.h

`main/include/common.h` 에는 다음의 내용이 정의되어있습니다.

- 사용할 헤더들에 대한 `#include` .
- 일부 변수에 대한 `#define` .
- Protocol에 대한 정의를 포함하는 `enum`

Pool

Simple NIDS 소스코드 중, `pool/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Pool의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `packet_pool.c` : Packet pool을 관리하는 함수들에 대한 구현
- `include/packet_pool.h` : Packet pool을 관리하는 함수들에 대한 선언 및 일부 값에 대한 정의.

packet_pool.c

`pool/packet_pool.c` 은 다음의 함수들을 구현합니다.

`int push_packet(struct packet_t *packet)`

Packet pool에 새로운 패킷을 push하는 기능을 합니다.

```
  if (packet_pool == NULL) {
          packet_pool = packet;
  } else {
          packet->next = packet_pool;
          packet_pool = packet;
  }
```

위의 코드는 packet pool에 새로운 `struct packet_t` 를 push할 때 실행되는 코드입니다. 기존의 packet pool이 비어있다면, packet을 추가합니다. 만일 packet pool이 비어있지 않다면, `next` 에 새로운 packet을 추가합니다. 해당 코드의 경우 Multithread로 동작하는 경우, critical section이 되기 때문에, `pthread_spin_lock` 와 `pthread_spin_unlock` 를 통해 mutual exclusion을 보장합니다. 또한 이는 사실상 stack처럼 동작하게 됩니다.

`struct packet_t* pop_packet()`

Packet pool에서 top을 pop 합니다. Packet pool이 stack으로 동작하기에, top의 packet 원소를 pop합니다.

```
  if (packet_pool == NULL) {
          packet = (struct packet_t *)malloc(sizeof(struct packet_t));
          memset(packet, 0, sizeof (struct packet_t));
  } else {
          packet = packet_pool;
          packet_pool = packet_pool->next;
          packet_items--;
  }
```

위의 코드는 packet pool 에서 `struct packet_t` 를 pop할 때 실행되는 코드입니다. `int push_packet(struct packet_t *packet)` 와 마찬가지로 critical section 이기에 `pthread_spin_lock` 와 `pthread_spin_unlock` 를 통해 mutual exclusion을 보장합니다. Pop의 경우, 만일 packet pool이 비어있는 상태라면, 공백의 `struct packet_t` 를 생성하여 반환합니다.

`int initialize_packet_pool()`

Packet pool을 초기화 하는 함수입니다.

```
for (i=0; i<INITIAL_ITEMS; i++) {
        struct packet_t *packet = (struct packet_t *)malloc(sizeof(struct
packet_t));
        push_packet(packet);
}
```

위의 코드를 통해 `INITIAL_ITEMS` 개의 `struct packet_t` 를 미리 할당해놓습니다. 이는 프로그램이 동작하며 `malloc` 을 매번 진행하여 실행 속도가 느려지는 것을 방지하기 위함힙니다. spinlock 또한 초기화 하기 때문에 `pthread_spin_init` 를 사용합니다.

`int destroy_packet_pool()`

생성된 packet pool의 모든 원소들을 free 하는 함수입니다.

```
while (packet != NULL) {
        struct packet_t *temp = packet;
        packet = packet->next;
        free(temp);
}
```

위의 코드를 통해 모든 packet을 `free` 합니다. 이후 `pthread_spin_destroy` 를 통해 spin lock을 삭제합니다.

packet_pool.h

`pool/include/packet_pool.h` 에는 다음의 내용이 정의되어있습니다.

- 일부 숫자에 대한 define (`INITIAL_ITEMS` , `MAXIMUM_ITEMS`)
- `packet_pool.c` 에서 구현된 함수들에 대한 선언.

Rule

Simple NIDS 소스코드 중, `rule/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Pool의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `rule_mgmt.c` : Simple NIDS의 탐지 규칙들을 읽어오고 등록하는 기능.
- `include/rule_mgmt.h` : 탐지 규칙에서 사용할 내용 정의.

rule_mgmt.c

`rule/rule_mgmt.c` 은 다음의 함수들을 구현합니다. 우선 모든 규칙들은 `rules.txt` 에 정의가 되어 있으며, 예시는 다음과 같습니다:

```
# IP rule
alert ip any any -> any 5000 (msg:"IP rule example";)
alert ip any any -> any any (msg:"IP rule example"; content:"payload";)
alert ip any any -> 221.163.205.3 any (msg:"specific IP example";
content:"boanlab"; regex:"^[0-9]*$";)
# TCP rule
alert tcp any any -> any any (msg:"TCP rule example"; content:"payload";)
alert tcp any any -> any 80 (msg:"TCP rule example"; content:"payload";
regex:"^[a-zA-Z]*$";)
# HTTP rule
alert http any any -> any any (msg:"HTTP rule example"; method:"GET";
content:"payload";)
alert http any any -> any any (msg:"HTTP data example"; content:"payload";
regex:"^[0-9]*$";)
# UDP rule
alert udp any any -> any any (msg:"UDP rule example"; content:"payload";)
alert udp any any -> any 53 (msg:"DNS request example"; content:"google";
regex:"^[a-zA-Z]*$";)
alert udp any 53 -> any any (msg:"DNS reply example"; content:"google";)
# ICMP rule
alert icmp any any -> any any (msg:"ICMP packet";)
alert icmp any 8 -> 8.8.8.8 0 (msg:"ping request example";)
alert icmp 8.8.8.8 0 -> any 0 (msg:"ping reply example";)
```

해당 내용을 올바르게 parsing 하기 위해서, 다음의 함수들을 사용합니다:

`void print_rule(struct rule_t *rule)`

`rule.txt` 에서 불러온 모든 `struct rule_t` 를 출력합니다.

`char * trim_left_space(char *str)`

어떤 문자열에서 왼쪽에 존재하는 whitespace를 제거합니다.

```
    alert icmp any any -> any any (msg:"ICMP packet";)
```

와 같이, 문자열이 처음에 로 시작하는 경우, 첫 문자가 나오기 전까지의 모든 whitespace를 제거합니다.

`char * remove_quotes(char *str)`

문자열에서 `""` 를 제거합니다. 예를 들어:

```
"ICMP packet"
```

을

```
ICMP packet
```

으로 변경합니다. 이는 `msg` 부분에 적힌 문자열 중 `"` 를 제외하여 필요한 문자열만을 포함하기 위함입니다.

`int load_rules(struct context_t *ctx)`

`rules.txt` 파일을 읽고 규칙들을 올바르게 로딩하는 함수입니다. 예를 들어 문자열이 다음과 같을 때, 이 함수는 다음의 순서로 규칙들을 로딩합니다.

```
alert ip any any -> 221.163.205.3 any (msg:"specific IP example";
content:"boanlab"; regex:"^[0-9]*$";)
```

1. 문자열을 각 ` ` 마다 구분합니다. 이는 각 요소들을 하나씩 잘라서 규칙을 얻기 위함입니다.

```
for (ptr = strtok(base, " "), idx = 0; ptr != NULL; ptr = strtok(NULL, "
"), idx = idx + 1) {
...
}
```

2. 문자열을 각 `;` 마다 구분합니다. 이는 각 문자열의 `(msg: ~~~~)` 부분의 구분이 되는 문자인 `;` 를 구별하여, 필터의 내용을 정하기 위함입니다.

```
for (ptr = strtok(options, ";"), idx = 0; ptr != NULL; ptr = strtok(NULL,
";"), idx = idx + 1) {
...
}
```

3. 문자열 입력에 따라, 올바르게 설정값을 저장합니다.

`int initialize_rule_table(struct context_t *ctx)`

Rule table을 이후 사용할 수 있도록 초기화합니다.

`int destroy_rule_table(struct context_t *ctx)`

모든 규칙이 적힌 rule table을 삭제합니다.

```
for (i=0; i<NUM_OF_PROTO; i++) {
        struct rule_t *rule = ctx->rule_table[i];
        while (rule != NULL) {
                struct rule_t *temp = rule;
                rule = rule->next;
                free(temp);
        }
}
free(ctx->rule_table);
```

위의 코드를 통해 각 `struct rule_t` 를 `free` 합니다.

Reception

Simple NIDS 소스코드 중, `reception/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Reception의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `pkt_live.c` : 실시간으로 패킷을 탐지하고, 탐지되는 패킷을 기반으로 침입을 확인하는 함수를 구현.
- `pkt_file.c` : 저장된 패킷 로그를 재생하여, 탐지되는 패킷을 통해 침입을 확인하는 함수를 구현.
- `include/pkt_live.h` : `pkt_live.c` 에서 구현된 함수에 대한 정의 및 필요한 헤더 파일 include 그리고 값을 define.
- `include/pkt_file.h` : `pkt_file.c` 에서 구현된 함수에서 사용하는 헤더 파일 include.

pkt_live.c

`lpcap` 에서 제공하는 기능들을 통해 실시간으로 패킷을 탐지합니다.

`int destroy_pkt_live()`

현재 동작중인 pcap handler 및 pcap의 loop을 올바르게 중지합니다.

`void packet_live_callback(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt)`

패킷이 pcap loop에서 탐지되었을 시, 호출되는 callback 함수입니다. 새로운 패킷이 들어왔을 경우, 다음의 절차를 순차적으로 실행합니다.

1. 이는 pcap에서 전달된 데이터를 Simple NIDS에서 사용하는 패킷 형식인 `struct packet_t` 에 맞추어서 decoding을 진행합니다 이후 packet pool에 새로 캡쳐된 패킷을 등록합니다. Decoding의 경우, Decoder에서 자세하게 설명 될 예정입니다.

```
if (local_ctx->decoder_func(local_ctx, packet) < 0) {
        push_packet(packet);
        return;
}
```

2. 이후 캡쳐된 패킷에서 사전에 등록한 규칙에 만족하는지를 탐지합니다. 이 탐지에 관한 내용은 이후 Detection에서 다시 언급이 됩니다.

```
struct rule_t *rule = local_ctx->string_match_func(local_ctx, packet);
```

3. 만일 탐지된 규칙이 있다면, log를 남깁니다.

`int pkt_live(struct context_t *ctx)`

pcap의 기능들을 사용하여 실시간으로 패킷을 캡쳐하는 함수입니다. 이 함수는 다음의 절차를 진행합니다:

1. 네트워크 인터페이스를 탐지하고, 인터페이스 중 `0` 번으로 등록된 인터페이스를 사용합니다.

```
if (pcap_findalldevs(&devices, ebuf) == -1) {
        printf("[ERROR] %s\n", ebuf);
        return -1;
}
```

2. pcap handler를 생성합니다.

```
handler = pcap_open_live(ctx->source, PCAP_SNAPSHOT, 1, PCAP_TIMEOUT,
ebuf);
```

3. pcap handler에 필요한 filter를 compile이후 적용시킵니다. Vagrant에서 SSH를 통해 프로그램을 실행시키므로, 22번으로 동작하는 모든 패킷은 무시합니다.

```
char filter[PCAP_FILTER_SIZE] = "! tcp port 22";
struct bpf_program filter_code;
if (pcap_compile(handler, &filter_code, filter, 0, 0) < 0) {
        printf("[ERROR] Failed to compile filter\n");
        return -1;
}
if (pcap_setfilter(handler, &filter_code) < 0) {
        printf("[ERROR] Failed to set filter\n");
        return -1;
}
```

4. pcap의 loop를 시작합니다.

```
pcap_loop(handler, 0, packet_live_callback, 0);
```

pkt_file.c

`pcap_live.c` 와 많은 기능을 공유하고 있습니다. pcap을 활용하여 실시간 패킷을 탐지하는 것이 아니라, 저장된 패킷 로그를 활용하여 탐지를 진행합니다.

`int destroy_pkt_file()`

현재 동작중인 pcap handler 및 pcap의 loop을 올바르게 중지합니다.

`void packet_file_callback(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt)`

패킷이 pcap loop에서 탐지되었을 시, 호출되는 callback 함수입니다. 패킷이 탐지되었을 시의 절차는 `packet_live_callback` 와 유사합니다.

`int pkt_file(struct context_t *ctx)`

패킷 로그 (사실상 dump)를 pcap을 통하여 읽어오는 것을 시작합니다.

1. pcap을 실시간으로 탐지하지 않고, 파일에서 읽어옵니다.

```
handler = pcap_open_offline(ctx->source, ebuf);
if (handler == NULL) {
        printf("[ERROR] %s\n", ebuf);
        return -1;
}
```

2. pcap loop를 활용하여 파일을 재생합니다

```
pcap_loop(handler, 0, packet_file_callback, 0);
```

pkt_file.h

`pkt_file.c` 에서 필요한 헤더 파일을 include 합니다.

pkt_live.h

`pkt_live.c` 에서 필요한 헤더파일을 include 하고, 필요한 값들을 define 합니다.

Decoder

Simple NIDS 소스코드 중, `decoder/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Decoder의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `decoder.c` : 실시간으로 패킷을 탐지하고, 탐지되는 패킷을 기반으로 침입을 확인하는 함수를 구현.
- `include/decoder.h` : `decoder.h` 에서 구현된 함수에서 사용하는 헤더 파일 include.

decoder.c

pcap에서 캡쳐한 패킷을 Simple NIDS에서 사용하는 `struct packet_t` 의 형태로 decoding 해주는 함수들을 가지고 있는 소스코드입니다.

`void mask_payload_printable(struct packet_t *packet)`

Payload의 데이터 중, `char` 로 표현할 수 없는 경우 `.` 으로 변환합니다.

```
        for (pi=0; pi<packet->payload_len; pi++) {
        if (packet->payload[pi] < 32 || packet->payload[pi] > 127) {
                packet->payload[pi] = '.';
        }
}
```

`int decoder(struct context_t *ctx, struct packet_t *packet)`

pcap에서 캡쳐한 패킷은 `struct packet_t` 의 `char pkt[MAX_PKT_SIZE]` 로 저장이 됩니다. 이로부터 포인터과 type casting 하여 필요한 내용을 추출합니다. 이를 달성하기 위해 다음의 절차를 수행합니다:

1. 패킷을 `struct ethhdr` 으로 type casting.

```
struct ethhdr *eth = (struct ethhdr *)packet->pkt;
```

2. Type casting 된 헤더의 protocol이 `ETH_P_IP` 인지 확인 후, ethernet header 만큼 포인터를 옮겨주어 IP header를 `struct iphdr` 로 저장

```
struct iphdr *iph = (struct iphdr *)(packet->pkt + sizeof(struct ethhdr));
```

3. Source, destination IP, 포트, MAC을 string으로 저장합니다. 이후 패킷이 TCP인지, UDP 인지, ICMP 인지 또는 일반 IPV4인지를 확인합니다.

```
if (iph->protocol == IPPROTO_TCP)
...
else  if (iph->protocol == IPPROTO_UDP)
...
else  if (iph->protocol == IPPROTO_ICMP)
...
else
```

각각 알맞는 헤더 형식으로 type casting을 진행하여, 헤더를 저장합니다. 에를 들어 TCP의 경우 다음처럼 type casting을 진행하고, 헤더를 `struct tcphdr` 형식으로 저장합니다:

```
struct tcphdr *tcph = (struct tcphdr *)(packet->pkt + sizeof(struct ethhdr)
+ (iph->ihl*4));
```

4. 추가적으로 각 프로토콜마다 처리할 것이 있다면 진행합니다. 예를 들어, HTTP를 탐지하는 경우 TCP의 헤더 중 내용을 파악해서 HTTP로 분류합니다.

```
if (packet->payload_len > 4) {
        if (packet->payload[0] == 'H' && packet->payload[1] == 'T' &&
packet->payload[2] == 'T' && packet->payload[3] == 'P') {
                packet->proto |= PROTO_HTTP;
        }
}
```

간단한 포인터 연산과 type casting을 통해, pcap에서 캡쳐한 패킷 정보를 올바르게 `struct packet_t` 의 형태로 구분할 수 있습니다.

decoder.h

`decoder.c` 에서 pcap의 패킷 정보를 typecasting 및 포인터 연산을 하기 위한 헤더들을 include 합니다.

Detection

Simple NIDS 소스코드 중, `detection/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Detection의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `match_regex.c` : 문자열을 RegEx표현을 통해 탐지하는 함수를 구현한 코드입니다.
- `match_string1.c` : 문자열에서 다른 문자열을 KMP 알고리즘을 통해 탐지하는 함수를 구현한 코드입니다.
- `match_string2.c` : 문자열에서 다른 문자열을 Boyer-Moore 알고리즘을 통해 탐지하는 함수를 구현한 코드입니다.
- `include/match_regex.h` : `match_regex.c` 에서 필요한 헤더를 include 하는 헤더 파일입니다.
- `include/match_string1.h` : `match_string1.c` 에서 필요한 헤더를 include 하는 헤더 파일입니다.
- `include/match_string2.h` : `match_string2.c` 에서 필요한 헤더를 include 하는 헤더 파일입니다.

match_regex.c

`<regex.h>` 를 include 하여 RegEx를 이용하여 캡쳐된 패킷의 payload를 탐지합니다. 이를 위해 `regcomp` , `regexec` , `regfree` 가 순차적으로 진행됩니다.

match_string1.c

문자열에서 다른 문자열을 KMP 알고리즘을 통해 탐지하는 함수를 구현한 코드입니다.

`static void make_skip_table(char *str, int len)`

KMP알고리즘에서 사용할 skip table을 생성하는 함수입니다.

`static int do_kmp(char *str, int slen, char *pattern, int plen)`

특정 문자열에서 다른 문자열 패턴을 KMP 알고리즘을 이용하여 검색하는 함수입니다.

`struct rule_t * match_string1(struct context_t *ctx, struct packet_t *packet)`

Decoder를 통해 생성된 `struct packet_t` 에서 KMP 알고리즘을 통해 문자열을 확인하는 함수입니다. 만일 기존에 설정한 규칙에 맞는 탐지가 이루어진 경우 해당하는 `struct rule_t` 를 반환합니다. 예를 들어, UDP의 경우 다음과 같습니다:

```
else if (packet->proto & PROTO_UDP) {
        rule = match_rules(ctx, UDP_RULES, packet);
        if (rule != NULL) {
                return rule;
        }
}
```

match_string1.c

문자열에서 다른 문자열을 Boyer-Moore 알고리즘을 통해 탐지하는 함수를 구현한 코드입니다.

`static int find(char *str, int len, char c)`

Boyer-Moore 알고리즘을 구현할 시 사용되는 find 함수입니다.

`static int do_boyer_moore(char *str, int slen, char *pattern, int plen)`

Boyer-Moore 알고리즘을 구현한 함수입니다.

`struct rule_t * match_string2(struct context_t *ctx, struct packet_t *packet)`

Decoder를 통해 생성된 `struct packet_t` 에서 KMP 알고리즘을 통해 문자열을 확인하는 함수입니다. 만일 기존에 설정한 규칙에 맞는 탐지가 이루어진 경우 해당하는 `struct rule_t` 를 반환합니다. 예를 들어, UDP의 경우 다음과 같습니다:

```
else if (packet->proto & PROTO_UDP) {
        rule = match_rules(ctx, UDP_RULES, packet);
        if (rule != NULL) {
                return rule;
```

```
        }
}
```

match_regex.h, match_string1.h, match_string2.h

> 모두 내용이 같은 소스코드라서 한번에 설명합니다.

필요한 헤더를 include하는 헤더 파일입니다.

Log

Simple NIDS 소스코드 중, `log/` 디렉토리의 소스코드에 대한 설명입니다.

Files

Log의 경우, 다음의 파일들을 포함하며 다음의 기능을 가지고 있습니다:

- `log_file.c` : 로그를 파일로 저장하는 기능을 가진 함수입니다.
- `log_stdout.c` : 로그를 stdout을 통해 출력하는 기능을 가진 함수입니다.
- `include/log_file.h` : `log_file.c` 에서 필요한 헤더를 include 하는 헤더 파일입니다.
- `include/log_stdout.h` : `log_stdout.c` 에서 필요한 헤더를 include 하는 헤더 파일입니다.

log_file.c

`int log_file(struct context_t *ctx, struct rule_t *rule, struct packet_t *packet)`

탐지된 `struct rule_t` 와, 원래의 패킷인 `struct packet_t` 에 따라서 문자열을 합성하는 기능을 가진 함수입니다. 문자열을 합성하기 위해 `sprintf` 등의 함수가 사용됩니다. Log로 남길 문자열이 합성된 경우, `fputs` 를 통해 문자열을 파일로 기록합니다.

```
FILE *fp = fopen(ctx->log, "a");

if (fp == NULL) {
        printf("%s", msg);
        return -1;
}

fputs(msg, fp);
```

log_stdout.c

`int log_stdout(struct context_t *ctx, struct rule_t *rule, struct packet_t *packet)`

탐지된 `struct rule_t` 와, 원래의 패킷인 `struct packet_t` 에 따라서 문자열을 출력하는 기능을 가진 함수입니다.

log_file.h, log_stdout.h

> 모두 내용이 같은 소스코드라서 한번에 설명합니다.

필요한 헤더를 include하는 헤더 파일입니다.

Limits

Simple NIDS의 한계점에 대해서 서술합니다.

다중 네트워크 인터페이스 동시 모니터링 불가

Simple NIDS의 경우, `lpcap` 을 하나의 thread로 하나의 네트워크 인터페이스만 모니터링합니다. 따라서 여러가지의 네트워크 인터페이스를 모니터링할 수 없습니다. 여러가지의 네트워크 인터페이스를 동시에 모니터링 하기 위해서는 각 reception 함수들을 multithread로 동작하게 해야합니다.

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.