PIOLINK

VECT 2.0 랜섬웨어

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

위협분석센터

VECT 2.0 랜섬웨어

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

• VECT 2.0 랜섬웨어 분석

1) 개요

VECT는 2025년 12월 러시아어권 사이버범죄 포럼에서 처음 확인된 서비스형 랜섬웨어(RaaS) 조직이다. 제휴자에게는 악성코드 제작 도구와 함께 피해자 협상 창구, 유출 자료 공개 사이트를 묶어 제공한다. 2026년 2월에는 Windows와 Linux, VMware ESXi를 지원하는 VECT 2.0을 공개했고, 3월에는 TeamPCP와의 협력을 발표하면서 BreachForums 등록 사용자를 제휴자로 받는 공개 가입 모델을 운영했다.

분석한 VECT 2.0은 관리자 권한을 확인한 뒤 복구 기능을 방해하고 서비스와 프로세스를 종료한다. 이어서 네트워크 공유와 Active Directory에 등록된 컴퓨터를 탐색해, Windows 원격 관리 기능인 WMI와 CIM으로 다른 시스템에서도 실행을 시도한다. 암호화를 마치면 랜섬노트를 남겨 자료를 이미 확보했다고 주장하지만, 주 실행 흐름에는 파일을 수집하거나 외부로 보내는 단계가 없다.

특히 128 KiB를 넘는 파일을 암호화하는 과정에서 특이점이 확인됐다. 각 파일을 네 구간으로 나눠 암호화하며 구간마다 새 일회용 난수값(난스, nonce)을 만드는데, 이 값을 담아 두는 공간이 하나뿐이라 마지막 난스인 N4만 파일 끝에 남는다. 파일을 정상적으로 복구하려면 N1~N3가 필요하지만 파일 어디에도 남지 않으므로, 고정 키와 N4로는 네 번째 구간만 복구할 수 있다.

2) 전체 실행 흐름

분석한 VECT 2.0은 기본 설정을 세우고 명령행 옵션을 적용한 뒤, 중복 실행 표시 파일을 확인한다. 이전 실행이 남긴 파일이 열리면 종료하고, 열리지 않으면 관리자 권한과 안전 모드 상태를 확인한다.

이어지는 단계는 확인한 조건에 따라 갈린다. 관리자 권한이 있으면 복구 기능을 방해한 뒤 암호화 대상 경로를 수집하고, 권한이 없으면 방해 단계를 건너뛴다. 경로 탐색과 파일 암호화는 별도 스레드에서 병행되며, 탐색 과정에서 랜섬노트를 생성한다. 로컬 암호화가 끝나면 시작 경로에 C: 또는 c:가 있을 때만 바탕화면 이미지를 만들고, 상세 출력(-v)이 켜져 있으면 최종 통계를 표시한다.

마지막으로 표시 파일을 기록한 뒤, --gpo가 켜져 있으면 원격 실행을 시도한다. 이후 흔적 삭제와 --stealth 조건에 따른 자기 삭제를 거쳐 종료한다.

< 그림 1 > VECT 2.0 전체 실행 흐름

3) 공격 기법 분석

항목	분석 결과
식별명	VECT 2.0
형식	PE32+ 실행 파일, Windows AMD64
크기	1,453,056 bytes
MD5	207b1a60f803d348c795d382f5aed9c3
SHA-256	8ee4ec425bc0d8db050d13bbff98f483fff020050d49f40c5055ca2b9f6b1c4d
내부 표시 문자열	VECT 2.0
생성 파일	.vect / !!!_READ_ME_!!!.txt / dvm3_wall.bmp

< 표 1 > 분석 대상 VECT 2.0 파일 정보

기본 옵션 초기화와 기본값

프로그램은 실행 옵션을 담아 두는 메모리 구조의 +336(0x150)부터 +339(0x153)까지 네 바이트를 모두 1로 채운 뒤, 이 영역에서 세 값을 읽어 기본 동작을 결정한다. +336은 --mount, +338은 --stealth, +339는 --gpo에 해당하며 세 옵션 모두 켜진 상태로 동작한다. 따라서 별도 지정 없이 실행해도 네트워크 공유 연결과 원격 실행, 자기 삭제 단계를 수행한다. 나머지 +337은 대응하는 옵션이 없으며 이후 어디에서도 참조하지 않는다.

```
v141 = GlobalMemoryStatusEx(Block);           // 물리 메모리 총량을 조회한다.
v142 = 15LL;
if ( v141 )
  v142 = Block[0].ullTotalPhys > 0x3FFFFFFFLL ? 20LL : 15LL;
*(a1 + 264) = v142;                           // 물리 메모리에 따라 15 또는 20을 설정 객체 +0x108에 저장한다.
std_string_replace_range(v166, 0LL, *(a1 + 280), byte_1401253F0, 0LL);// 설정 문자열(+272)을 빈 문자열로 초기화한다(원본 전역 바이트의 첫 값은 NUL).
std_string_replace_range(v167, 0LL, *(a1 + 312), byte_1401253F0, 0LL);// 설정 문자열(+304)을 빈 문자열로 초기화한다(원본 전역 바이트의 첫 값은 NUL).
v143 = _mm_loadu_si128(&xmmword_140125500);
LOBYTE(Block[0].ullTotalVirtual) = -13;
*(a1 + 336) = 0x1010101;                      // 설정 플래그 +336~+339를 모두 1로 초기화한다.
```

< 그림 2 > 기본 옵션 4바이트 초기값

```
v97 = *(&qword_14011F6E0 + 338);              // 내장 설정 객체 338의 값을 자기 삭제(--stealth) 플래그의 초기값으로 읽는다.
v93 = *(&qword_14011F6E0 + 339);              // 내장 설정 객체 +339의 값을 원격 실행(--gpo) 플래그의 초기값으로 읽는다.
v95 = *(&qword_14011F6E0 + 336);              // 설정 +336을 --mount 기본값으로 읽는다.
```

< 그림 3 > 기본 설정 활성화

명령행 옵션과 실행 범위

지원하는 옵션은 도움말, 상세 출력, 대상 경로, 자격 증명, 공유 연결, 원격 실행, 자기 삭제, 안전 모드 설정이다.

이 가운데 --gpo와 --mount, --stealth는 기본으로 켜져 있어 --no-gpo, --no-mount, --no-stealth로 꺼야 한다.

경로 옵션(-p, --path)에 값을 주면 자동으로 찾은 드라이브 목록 대신 지정한 경로만 암호화한다. 하지만 네트워크 공유 연결은 암호화 경로를 정하기 전에 실행되기 때문에 관리자 권한에서 --mount가 켜져 있으면 경로를 지정해도 공유 연결을 시도한다.

```
if ( !std_string_compare_cstr(Block, "-h") || !std_string_compare_cstr(Block, "--help") )
  break;                                // 일치하면 인자 파싱을 끝내고 콘솔을 연 뒤 도움말을 호출하고 종료한다.
if ( !std_string_compare_cstr(Block, "--stealth") )// 자기 삭제 플래그를 1로 설정한다.
{
  v97 = 1;
  goto LABEL_4;
}
if ( !std_string_compare_cstr(Block, "--no-stealth") )// 자기 삭제 플래그를 0으로 설정한다.
{
  v97 = 0;
  goto LABEL_4;
}
if ( !std_string_compare_cstr(Block, "-v") || !std_string_compare_cstr(Block, "--verbose") )// 상세 출력 플래그를 1로 설정해 콘솔 생성, 500ms 진행 출력, 최종 통계 출력을 활성화한다.
{
  v98 = 1;
  goto LABEL_4;
}
if ( std_string_compare_cstr(Block, "-c") && std_string_compare_cstr(Block, "--creds") )// 다음 argv의 표준 Base64 문자열을 저장한다.
{                                       // 다음 argv을 암호화 대상 경로로 저장한다.
  if ( std_string_compare_cstr(Block, "-p") && std_string_compare_cstr(Block, "--path") )
  {                                     // 원격 실행 플래그를 1로 설정한다.
    if ( std_string_compare_cstr(Block, "--gpo") )
    {                                   // 원격 실행 플래그를 0으로 설정해 원격 실행 스레드 3개의 생성을 막는다.
      if ( std_string_compare_cstr(Block, "--no-gpo") )
      {                                 // 볼륨 문자 배정과 네트워크 공유 연결 블록을 활성화한다.
        if ( std_string_compare_cstr(Block, "--mount") )
        {                               // 볼륨 문자 배정과 네트워크 공유 연결 블록을 비활성화한다.
          if ( std_string_compare_cstr(Block, "--no-mount") )
          {
            v92 = v96;
            if ( !std_string_compare_cstr(Block, "--force-safemode") )// 강제 SafeBoot 플래그를 1로 설정한다.
```

< 그림 4 > 명령행 인자 설정

```
Console_Printf_CLI("\n  VECT 2.0\n\n");       // VECT 2.0 배너를 출력한다.
Console_Printf_CLI("  -h, --help          Help\n");
Console_Printf_CLI("  -v, --verbose       Verbose output\n");
Console_Printf_CLI("  -p, --path <dir>    Target specific path\n");
Console_Printf_CLI("  -c, --creds <b64>   Override credentials\n");// -c/--creds <b64>가 기본 자격증명을 덮어쓰는 Base64 입력 옵션이라고 출력한다.
Console_Printf_CLI("  --gpo               Enable GPO spread (default: on)\n");
Console_Printf_CLI("  --no-gpo            Disable GPO spread\n");// 도움말은 --gpo를 기본 활성, --no-gpo를 비활성으로 표시한다.
Console_Printf_CLI("  --mount             Enable network mount (default: on)\n");
Console_Printf_CLI("  --no-mount          Disable network mount\n");// 도움말은 --mount를 기본 활성, --no-mount를 비활성으로 표시한다.
Console_Printf_CLI("  --stealth           Enable self-delete (default: on)\n");
Console_Printf_CLI("  --no-stealth        Disable self-delete\n");// --stealth로 자기 삭제 사용 여부를 제어한다고 출력한다.
return Console_Printf_CLI("  --force-safemode    Force safemode boot\n\n");// --force-safemode가 안전 모드 부팅 설정을 강제하는 옵션이라고 출력한다.
```

< 그림 5 > 도움말에 표시되는 전체 옵션

자격 증명

자격 증명은 파일 안에 Base64로 들어 있으며, 디코딩하면 <도메인>₩<사용자명>:<암호> 형식이라 이를 셋으로 나눠 사용하지만 -c 옵션으로 다른 값을 넣을 수도 있다.

이 자격 증명이 쓰이는 곳은 WMI와 원격 관리 규격인 CIM을 이용한 원격 실행뿐이며, 파일 복사와 공유 연결은 현재 로그인 계정 권한으로 시도한다.

```
j_j_free_4(v7);                         // 입력 디코딩이 끝나면 1024바이트 임시 테이블을 해제한다.
v19 = sub_14007EE20(&v42, 58, 0LL);     // 디코딩 결과에서 첫 번째 ':' 구분자를 찾는다.
if ( v19 == -1LL )                      // 콜론이 없으면 실패한다.
```

< 그림 6 > Base64 문자열에서 도메인, 계정, 암호 분리

중복 실행 방지를 위한 표시 파일 확인

중복 실행 여부는 C:₩ProgramData₩.vect를 읽기 전용으로 열 수 있는지만 보고 판단하며, 파일이 열리면 이전 실행이 남긴 표시로 보고 종료한다.

표시 파일은 실행 초기에 확인하지만, 로컬 암호화와 랜섬노트, 바탕화면 처리가 모두 끝난 뒤에야 같은 경로의 파일에 문자 '1'을 기록하고, 상세 출력(-v)이 켜져 있으면 기록 직전에 최종 통계를 표시한다. 확인과 기록 사이가 이렇게 벌어져 있어, 두 프로세스가 거의 동시에 시작하면 양쪽 모두 표시 파일이 없다고 판단해 함께 실행될 수 있다.

```
std_string_from_cstr(v110, v19);          // C:\ProgramData\.vect 표시 경로를 문자열로 만든다.
StdIfstream_OpenPathA(Block, v110, 8);    // C:\ProgramData\.vect를 읽기 모드로 열어 성공 여부만 본다.
if ( v122 )                               // 표시 파일이 열리면 재실행으로 보고 후속 동작을 중단한다.
```

< 그림 7 > 표시 파일을 이용한 중복 실행 확인

```
v121 = 0;                                 // 런타임 컨텍스트 +0x38을 원자적 xchg로 0으로 바꿔 500ms 진행률 반복의 종료를 요청한다.
if ( v101 )                               // 진행률 스레드가 있으면 종료까지 기다린다.
  StdThread_JoinAndClear(&v101);          // 앞에서 컨텍스트 +0x38을 0으로 내린 뒤 진행률 스레드를 join하고 핸들을 비운다.
if ( v98 )                                // verbose 플래그가 0이면 PrintFinalReport를 건너뛴다.
  PrintFinalReport(Block);                // verbose 모드이면 PrintFinalReport를 호출한다.
StdOfstream_OpenPathA(&v116, v110, 16); // 후처리 뒤 C:\ProgramData\.vect 표식 파일 생성을 시도한다.
if ( (v118 & 5) == 0 )                    // 쓰기 전에 출력 스트림의 fail/bad 비트(0x5)가 없으면 한 바이트 문자열 1을 기록한다.
  StdOstream_InsertCharsPadded(v116.m128i_i64, "1", 1LL);
v105 = 0LL;
*v104 = 0LL;
if ( v93 )                                // --gpo가 켜지면 원격 실행 스레드 3개를 준비한다.
{                                         // --gpo 모드 정보 출력
```

< 그림 8 > 로컬 처리 후 최종 통계, 표시 파일 기록

권한 확인

Windows 토큰 정보를 읽어 현재 프로세스가 관리자 권한인지 확인한다. 다만 권한이 부족해도 스스로 끌어올리려 시도하지 않고, 확인한 권한 수준에 맞춰 이후 동작만 나눈다.

```
v0 = 0;                                   // 실패와 비상승 상태의 공통 반환값을 0으로 초기화한다.
TokenHandle = 0LL;
CurrentProcess = GetCurrentProcess();     // 의사 핸들을 가져와 별도 핸들 정리 없이 토큰 열기에 사용한다.
if ( !OpenProcessToken(CurrentProcess, 8u, &TokenHandle) )// TOKEN_QUERY(0x8) 권한으로 현재 프로세스 토큰을 연다.
  return 0LL;
ReturnLength = 4;                         // TOKEN_ELEVATION 구조체 크기인 4바이트를 지정한다.
if ( GetTokenInformation(TokenHandle, TokenElevation, &TokenInformation, 4u, &ReturnLength) )// TokenElevation으로 현재 토큰의 상승 여부를 조회한다.
  LOBYTE(v0) = TokenInformation != 0;     // 조회가 성공한 경우 TokenIsElevated 정수 필드가 0이 아니면 1로 변환한다.
CloseHandle(TokenHandle);                 // 토큰을 연 뒤에는 정보 조회 성공 여부와 무관하게 토큰 핸들을 닫는다.
return v0;                                // 상승 상태만 1이며, 그 밖의 상태와 API 오류는 모두 0으로 합쳐진다.
```

< 그림 9 > Windows 토큰 정보로 관리자 권한 확인

안전 모드 설정과 해제

안전 모드 여부는 GetSystemMetrics로 부팅 상태(SM_CLEANBOOT)를 조회하며, 반환값이 0이 아니면 안전 모드로 판단한다. --force-safemode가 켜지면 bcdedit를 창 없이 실행해 다음 부팅을 최소 안전 모드로 설정하고 최대 5초간 기다린 뒤, 재부팅하지 않고 다음 단계로 넘어간다.

관리자 권한으로 안전 모드에서 실행된 경우, 종료 단계에서 부팅 구성 데이터(BCD)의 safeboot 설정을 지운다. SafeBoot와 Run 레지스트리 값은 그대로 남아 다음 부팅에도 적용된다.

```
if ( !v100 && v96 )                     // 관리자 권한에서 --force-safemode가 켜지면 안전 모드 설정을 시도한다.
  RunBcdedit SetSafeBootMinimal();      // bcdedit로 기본 부팅 항목의 안전 모드 설정을 시도한다.
```

< 그림 10 > bcdedit 안전 모드 설정 시도

재부팅 후 실행을 위한 등록과 보안 설정 변경

관리자 권한이면 --force-safemode 사용 여부와 무관하게 현재 실행 파일 경로를 SafeBoot₩Minimal과 SafeBoot₩Network, HKLM₩Run에 등록한다. SafeBoot와 HKLM₩Run에 쓰는 등록값 이름은 실행 파일명에서 처음 나오는 소문자 '.exe' 앞부분이고, 소문자 '.exe'가 없으면 파일명 전체를 쓴다.

이어서 PowerShell의 Set-MpPreference로 실시간 감시와 동작 감시, 다운로드 파일 검사(IOAV), 스크립트 검사를 끄도록 요청하고 DisableTaskMgr=1을 기록해 작업 관리자 실행을 막는다. 종료 단계에서 되돌리는 항목은 BCD의 safeboot 설정 하나뿐이다. SafeBoot와 Run 등록, Defender 설정, 작업 관리자 차단은 실행이 끝난 뒤에도 남는다.

```
GetModuleFileNameA(0LL, Block, 0x104u);// GetModuleFileNameA(NULL, ..., 260)으로 현재 실행 파일의 전체 경로를 얻는다.
std_string_from_cstr(Stream, Block);  // GetModuleFileNameA가 반환한 char 경로를 std::string으로 복사한다.
AttemptRegisterPersistence_SafeBootAndRun(Stream);// SafeBoot 허용 목록과 HKLM Run 등록을 시도한다.
```

< 그림 11 > SafeBoot와 Run 레지스트리 등록

```
std_string_destroy(Stream);           // 현재 실행 파일 경로를 담은 std::string을 소멸한다.
AttemptDisableDefender_PowerShell();  // PowerShell로 Microsoft Defender 검사 비활성화를 시도한다.
AttemptStopServices_TerminateProcesses_DisableTaskMgr();// 상승 토큰일 때 서비스 중지 제어, 대상 프로세스 종료, 현재 사용자 작업 관리자 정책 쓰기를 시도한다.
```

< 그림 12 > Defender 보호 설정 변경과 작업 관리자 차단

파일 암호화 전 서비스 및 프로세스 종료

관리자 권한이면 Windows 서비스 제어 관리자(SCM)를 통해 28개 서비스 이름을 차례로 조회하고, 실행 중인 항목에 중지 명령을 보낸다. 프로세스는 실행 파일명이 8개 대상 이름 중 하나와 대소문자 구분 없이 정확히 일치할 때 종료 코드 9를 지정해 강제 종료를 요청한다.

```
v4 = OpenServiceA(v0, *v3, 0x2Cu);     // 서비스 조회·중지 권한으로 대상 서비스를 연다.
if ( v4 )                              // 해당 정확 이름의 서비스가 없거나 접근이 거부되면 로그 없이 다음 항목으로 넘어간다.
{                                      // 36바이트 SERVICE_STATUS_PROCESS 정보를 요청한다.
  if ( QueryServiceStatusEx(v4, SC_STATUS_PROCESS_INFO, Buffer, 0x24u, &pcbBytesNeeded)
    && (Buffer[0].dwCurrentState & 0xFFFFFFFD) != 1 )
  {
    ControlService(v4, 1u, Buffer);    // STOPPED·STOP_PENDING이 아니면 서비스 중지를 요청한다.
  }
  CloseServiceHandle(v4);              // 상태 조회나 중지 요청 결과와 무관하게 현재 서비스 핸들을 닫고 다음 이름으로 진행한다.
```

< 그림 13 > Windows 서비스 제어 관리자를 이용한 중지 요청

```
RequestStopServicesInList_ViaSCM();          // 28개 정확 서비스명을 대상으로 SERVICE_CONTROL_STOP을 시도한다.
AttemptTerminateProcessesByExactName_Toolhelp();// 이름이 일치하는 8개 대상 프로세스의 종료를 시도한다.
result = IsProcessElevated();                // 서비스/프로세스 방해 작업 뒤 상승 상태를 다시 검사한다.
if ( result )
  return Registry_DisableTaskManager();      // Registry_DisableTaskManager의 반환값을 그대로 전달한다.
```

< 그림 14 > 대상 파일명 일치 시 프로세스 종료 요청

Windows 복구용 볼륨 섀도 복사본 삭제

같은 관리자 권한 조건에서 vssadmin delete shadows /all /quiet를 창 없이 실행하고 최대 30초를 기다린다. 30초 안에 종료되지 않으면 vssadmin 프로세스를 강제로 종료하지 않은 채 다음 단계로 넘어간다. 따라서 해당 프로세스가 백그라운드에서 계속 실행될 수 있다.

```
if ( v2[2].m128i_i8[4] )                     // 명령 뒤의 플래그가 1인 경우에만 반복 XOR을 수행한다.
{                                            // `0xCB7BFF79F115AB07`을 반복 적용한 정확한 36바이트 결과는 `vssadmin delete shadows /all /quiet\0`이다.
  for ( i = 0LL; i != 36; ++i )
    v2->m128i_i8[i] ^= 0xCB7BFF79F115AB07uLL >> (8 * (i & 7u));
  v2[2].m128i_i8[4] = 0;                     // 플래그를 0으로 바꾸어 다음 호출부터는 평문화된 전역 버퍼를 재사용한다.
}
Source[0] = v12;                             // 평문 명령을 SSO 기반 `std::string`에 복사한다.
```

< 그림 15 > vssadmin을 이용한 볼륨 섀도 복사본 전체 삭제 시도

종료 대상 서비스 및 프로세스

아래 표는 설정에서 확인된 종료 대상을 정리한 것이다. 서비스 28개, 프로세스 8개이며 백업, 데이터베이스, 문서 편집기, 메일 클라이언트 관련 항목이 함께 들어 있다.

구분	대상 이름	처리 방식
서비스 28개	vss, sql, svc$, memtas, mepocs, sophos, veeam, backup, GxVss, GxBlr, GxFWD, GxCVD, GxCIMgr, DefWatch, ccEvtMgr, ccSetMgr, SavRoam, RTVscan, QBFCService, QBIDPService, Intuit.QuickBooks.FCS, QBCFMonitorService, YooBackup, YooIT, MSSQLSERVER, MySQL57, MySQL80, wbengine	SCM에서 대상 서비스를 이름으로 조회한 뒤, 실행 중인 경우 중지 명령을 보내고 다음 항목으로 넘어간다.
프로세스 8개	sql.exe, oracle.exe, excel.exe, outlook.exe, winword.exe, mysqld.exe, firefox.exe, thunderbird.exe	실행 파일명을 대소문자 구분 없이 정확히 비교하고, 대상 이름과 일치하는 프로세스에 종료를 요청한다.

< 표 2 > 종료 대상 서비스와 프로세스

로컬 볼륨과 암호화 대상 경로 수집

관리자 권한에서 --mount가 켜져 있으면 드라이브 문자가 없는 볼륨을 찾아 X:₩부터 C:₩ 방향으로 사용 가능한 드라이브 문자를 할당한다. 이 단계는 --path 적용보다 앞에 있어, 경로를 지정해도 그대로 실행된다.

암호화 대상 경로는 이동식 디스크와 고정 디스크, 원격 드라이브, RAM 디스크, 그리고 Windows가 인식한 네트워크 공유에서 모으며 CD-ROM은 제외한다.

```
FirstVolumeW = FindFirstVolumeW(v10, 0x8000u);// FindFirstVolumeW로 첫 볼륨 GUID를 가져와 열거를 시작한다.
if ( FirstVolumeW != -1LL )
{
  do
  {
    if ( !v1 )
      break;
    while ( GetVolumePathNamesForVolumeNameW(v10, v9, 0x78u, &v6) && wcslen(v9) == 3 )// 첫 반환 경로 길이가 3이면 드라이브 문자가 있다고 본다.
    {
      if ( !FindNextVolumeW(FirstVolumeW, v10, 0x8000u) )
        goto LABEL_12;
    }
    v5 = v8[--v1];
    SetVolumeMountPointW(v5, v10);            // 후보 드라이브 문자를 현재 볼륨 GUID에 연결한다.
  }
  while ( FindNextVolumeW(FirstVolumeW, v10, 0x8000u) );// FindNextVolumeW로 다음 볼륨 GUID를 요청하고 성공하는 동안 같은 처리를 반복한다.
LABEL_12:
  FindVolumeClose(FirstVolumeW);              // FindVolumeClose로 볼륨 열거 핸들을 닫는다.
```

< 그림 16 > 볼륨 GUID와 드라이브 문자 할당 시도

```
LogicalDrives = GetLogicalDrives();           // GetLogicalDrives로 A:~Z: 사용 여부를 확인한다.
v2 = 65;
do
{
  while ( 1 )
  {
    if ( (LogicalDrives & 1) != 0 )
    {
      wcscpy_s(Destination, 8uLL, L"X:\\");
      Destination[0] = v2;
      DriveTypeW = GetDriveTypeW(Destination);
      if ( DriveTypeW - 2 <= 2 || DriveTypeW == 6 )// 이동식·고정·원격·RAM 드라이브만 대상에 추가한다.
      {
        WideCharToMultiByte(0xFDE9u, 0, Destination, -1, MultiByteStr, 8, 0LL, 0LL);
        Block[0] = v37;
        v4 = strlen(MultiByteStr);
        sub_1400F6330(Block, MultiByteStr, &MultiByteStr[v4]);
        sub_1400F3DE0(a1, Block);
        if ( Block[0] != v37 )
          break;
      }
    }
    ++v2;
    LogicalDrives >>= 1;
    if ( v2 == 91 )
      goto LABEL_8;
  }
  ++v2;
  LogicalDrives >>= 1;
  j_j_free_4(Block[0]);
}
while ( v2 != 91 );
LABEL_8:
if ( !WNetOpenEnumW(1u, 1u, 0, 0LL, &hEnum) ) // 현재 연결된 네트워크 디스크를 열거한다.
```

< 그림 17 > Windows API를 이용한 로컬 드라이브 목록 수집

네트워크 공유 연결

공유 연결도 관리자 권한에서 --mount가 켜져 있을 때 동작한다. 현재 로그인 계정 권한으로 Z:에서 M: 방향으로 사용 가능한 드라이브 문자를 찾아 연결을 시도한다. 앞서 본 X:~C: 방향의 로컬 볼륨 문자 할당과는 별개 단계다.

```
v80 = WNetAddConnection2A(&WideCharStr, 0LL, 0LL, 4u);// 후보 UNC 공유를 임시 드라이브 문자로 연결한다.
v81 = *MultiByteStr;
if ( !v80 || v80 == 85 )                    // 성공 또는 이미 할당됨이면 다음 드라이브 문자로 이동한다.
{                                           // 연결 성공 시 원격 경로와 드라이브 문자를 로그로 출력한다.
  if ( byte_14011F870 )
  {
    Console_Printf_Network("[NET] Mounted %s -> %s\n", *(v73 + 32), *MultiByteStr);
    v81 = *MultiByteStr;
  }
  --v74;
}
```

< 그림 18 > 현재 로그인 계정 권한을 이용한 네트워크 공유 연결

UNC 중복 제거

수집한 경로 중 같은 네트워크 공유를 가리키는 UNC 경로가 여러 번 확인되면 하나만 남기고 나머지를 제거한다. 드라이브 문자로 이미 매핑된 공유는 목록에 추가하지 않으며, 매핑되지 않은 UNC 경로만 암호화 대상에 넣는다. 이 과정을 거친 뒤에도 대상 경로가 하나도 없으면 C:₩를 사용한다.

```
    if ( !WNetGetConnectionW(MultiByteStr, Block, &nLength) && !wcsicmp(Block, *(i + 32)) )
      break;                                // 이미 드라이브 문자로 매핑된 UNC 경로는 대상 목록에 중복 추가하지 않는다.
    if ( ++v22 == 91 )
    {
      Block[0] = v37;
      v24 = strlen(Destination);
      sub_1400F6330(Block, Destination, Destination + v24);
      sub_1400F3DE0(a1, Block);             // 드라이브 문자에 매핑되지 않은 UNC 경로를 암호화 대상 루트 목록에 추가한다.
      if ( Block[0] != v37 )
        j_j_free_4(Block[0]);
      if ( byte_14011F860 )
        Log_FoundUNCPath("[*] Found UNC path: %s\n", Destination);
      break;
    }
  }
}
if ( *a1 == *(a1 + 8) )                     // 필터를 거친 대상 루트가 하나도 없으면 문자열 'C:\\'을 목록에 추가한다.
```

< 그림 19 > 드라이브 문자에 매핑된 UNC 경로 중복 제거

디렉터리와 파일 제외 조건

제외 판정은 디렉터리와 파일을 따로 한다. 디렉터리는 이름이 비어 있거나 $로 시작하는 경우, 또는 시스템, 부팅, 프로그램 관련 12개 제외 대상 이름과 일치하는 경우 대소문자를 구분하지 않고 제외한다.

파일 쪽은 부팅 관련 파일명과 .exe, .dll, .sys 확장자를 제외한다. 랜섬노트는 !!!_READ_ME_!!!.txt와 정확히 일치할 때만 제외하므로 대소문자가 다르면 암호화 대상에 들어간다. 파일명이 .vect로 끝나고 .vect 앞에 한 글자 이상 있으면 이미 암호화된 파일로 보고 제외한다. 반면 파일명이 정확히 .vect이면 제외 조건에 해당하지 않는다. 설치 패키지(.msi)를 걸러내는 조건은 없다. 심볼릭 링크와 디렉터리 연결 지점을 일반 디렉터리처럼 처리하므로, 연결된 경로까지 탐색할 수 있다.

```
wcscpy_s(v29, 0x8000uLL, v28);              // 현재 디렉터리 경로 뒤에 \\*를 붙여 FindFirstFileW 열거 패턴을 만든다.
wcscat_s(v29, 0x8000uLL, L"\\*");           // '<디렉터리>\\*' 형태의 FindFirstFileW 검색 경로를 만든다.
if ( byte_14011F860 && __ROR4__(-1030792151 * v19, 2) <= 0x28F5C28u )// 처리 경로 수가 100의 배수일 때 누적 스캔 경로 수와 큐에 넣은 파일 수를 출력한다.
  Console_Printf_Progress("[Scanner] Scanned %d paths, queued %d files\n", v19, v20);
FirstFileW = FindFirstFileW(v29, &v27);     // 구성한 <디렉터리>\\* 패턴으로 FindFirstFileW를 호출해 첫 항목을 얻는다.
v14 = Source + 5;                           // 스레드 인자 구조체 +0x0A의 64개 wchar_t 버퍼를 참조한다.
if ( FirstFileW != -1LL )
{
  v17 = v2;
  while ( (v27.dwFileAttributes & 0x10) == 0 )// FILE_ATTRIBUTE_DIRECTORY(0x10)만 검사한다.
  {                                         // 시스템 파일명·exe/dll/sys·노트명·.vect 접미사를 검사한다.
    if ( FileFilter_ShouldSkip(v27.cFileName)
      || !wcscmp(v27.cFileName, v14)
      || (v10 = wcslen(v27.cFileName), v11 = wcslen(v18), v10 > v11) && !wcsicmp(&v27.cFileName[v10 - v11], v18) )
```

< 그림 20 > 파일 암호화 제외 조건

하위 경로 탐색과 암호화 작업 분리

탐색과 암호화는 서로 다른 스레드가 맡는다. 경로 탐색 스레드가 시작 경로 아래를 훑으면서 조건을 통과한 파일을 대기 목록에 넣으면, 암호화 스레드가 목록에서 하나씩 꺼내 처리한다. 랜섬노트는 탐색한 디렉터리에 같은 파일이 없을 때만 만든다.

처리 건수는 암호화 함수를 호출한 직후 올라간다. 실패한 파일도 여기에 포함된다.

```
wcscpy_s(v29, 0x8000uLL, v28);          // <dir>/<filename> 전체 경로 생성.
wcscat_s(v29, 0x8000uLL, asc_1401255C6);
wcscat_s(v29, 0x8000uLL, v27.cFileName);
Queue_PushBlocking(v16 + 544, v29, 1);// 필터를 통과한 파일 경로를 변환 큐에 넣는다.
++v20;
if ( !FindNextFileW(FirstFileW, &v27) )
{
```

< 그림 21 > 조건을 통과한 파일을 암호화 대기 목록에 추가

```
v4 = Queue_PopBlocking(v2, 1, &v7);         // 변환 큐에서 다음 파일 경로를 기다린다.
v5 = v4;
if ( !v4 )
  break;
if ( byte_14011F860 )                       // verbose이면 처리 중인 UTF-8 경로를 출력한다.
{
  WideCharToMultiByte(0xFDE9u, 0, v4, -1, MultiByteStr, 512, 0LL, 0LL);
  Console_Printf_Progress("[*] %s\n", MultiByteStr);
}
EncryptFile_ReadTransformWrite(v1, v5);     // 큐에서 받은 파일 경로의 변환을 시도한다.
_InterlockedAdd64(v1, 1uLL);                // 변환 결과와 무관하게 처리 시도 건수를 1 증가시킨다.
ProcessHeap = GetProcessHeap();
HeapFree(ProcessHeap, 0, v5);               // 암호화 큐에서 꺼낸 경로 문자열의 힙 버퍼를 해제한다.

eturn 0LL;                                  // 큐에서 NULL 종료 표식을 받으면 0을 반환한다.
```

< 그림 22 > 암호화 호출 직후 증가하는 처리 건수

논리 프로세서 수에 따른 작업 스레드 수

작업 스레드 수는 논리 프로세서 수에 따라 정해진다. 프로세서가 2개 이하면 16개, 3~4개면 8배, 5~8개면 6배, 9개 이상이면 4배이며 상한은 256개다. 이 중 약 1/8은 경로 탐색에, 나머지는 파일 암호화에 배정한다. 계산 결과가 적더라도 경로 탐색은 최소 4개, 암호화는 최소 12개를 만든다.

문제는 이렇게 만든 스레드가 모두 끝나기를 기다리는 단계에서 나타난다. 여러 스레드의 종료를 한 번에 기다리는 Windows API는 최대 64개까지만 처리하는데, 논리 프로세서가 19개 이상이면 암호화 스레드가 이 한도를 넘는다. 코드는 스레드 목록을 나누지 않고 한 번에 전달하므로, 고사양 환경에서는 종료를 기다리는 단계가 실패할 수 있다.

```
if ( byte_14011F860 )                           // 스캐너 완료 후 암호화 스레드 대기 메시지를 출력한다.
  Console_Printf_Progress("[*] Scanners complete. Waiting for encryptors...\n");
v31 = lpHandles[1];
v32 = lpHandles[0];
v33 = lpHandles[1] - lpHandles[0];
if ( v33 )
{                                               // 암호화 스레드 수만큼 암호화 큐(a1+544)에 NULL을 넣어 모든 암호화 스레드가 자연 종료되게 한다.
  for ( j = 0LL; j != v33; ++j )
    Queue_PushBlocking(a1 + 544, 0LL, 1);
}
if ( v31 != v32 )
  WaitForMultipleObjects(v33, v32, 1, 0xFFFFFFFF);// 64개를 넘을 수 있는 암호화 스레드 핸들을 한 번에 대기한다.
if ( byte_14011F860 )                           // 암호화 스레드 완료 메시지를 출력한다.
  Console_Printf_Progress("[*] Encryptors complete.\n");
```

< 그림 23 > 암호화 스레드 대기 요청

고정 키와 구간별 난스 생성

암호화 키는 실행 파일 안에 32바이트 고정값으로 저장돼 있다. 구간마다 Windows 난수 생성기로 12바이트 난스를 새로 만들어 ChaCha20-IETF로 내용을 암호화한다. 위변조를 확인하는 인증값(태그)은 붙이지 않는다.

128 KiB를 넘는 파일은 네 위치에서 데이터를 읽고, 읽기에 성공한 구간마다 새 난스를 생성해 암호화한다. 네 구간이 모두 처리되면 N1부터 N4까지 차례로 만들어진다.

```
v7 = state + 64;                                // 고정 32바이트 키는 상태 구조체 +64에 있다.
Crypto_GenerateRandomBytes(&v9, 0xCuLL);        // RtlGenRandom으로 12바이트 난스를 생성한다.
*nonce_out = v9;                                // 난스의 앞 8바이트를 출력 버퍼에 복사한다.
*(nonce_out + 2) = v10;                         // 나머지 4바이트를 복사해 12바이트 난스를 완성한다.
return ChaCha20_IETF_Xor(buffer, buffer, length, &v9, v7);// 같은 버퍼를 입력·출력으로 사용해 ChaCha20-IETF 변환한다.
```

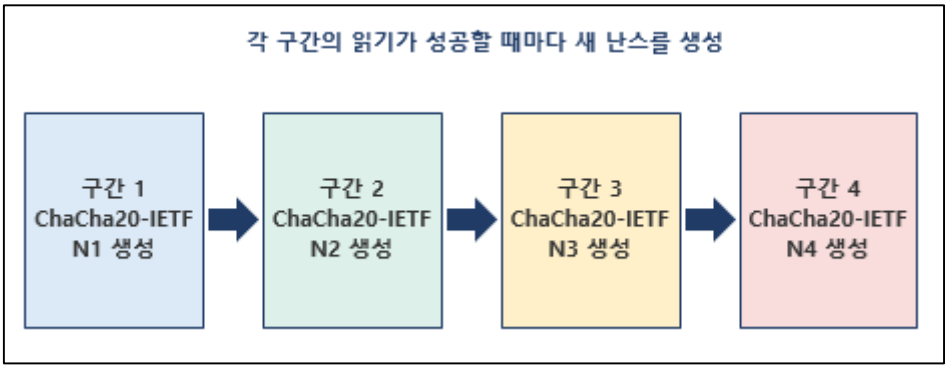
< 그림 24 > 난스 생성과 ChaCha20-IETF 암호화

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

대용량 파일의 네 구간 암호화

파일 크기를 N이라 하면, 128 KiB를 넘는 파일은 0, N/4, 2N/4, 3N/4 위치에서 각각 최대 32 KiB를 암호화한다. 해당 위치에서 1바이트라도 읽혀야 그 구간을 처리한다. 네 구간이 모두 32 KiB씩 처리되더라도 실제로 바뀌는 데이터는 최대 128 KiB이고, 구간 사이 영역은 원본 그대로 남는다.

< 그림 25 > 128 KiB 초과 파일의 네 암호화 위치와 최대 처리 범위

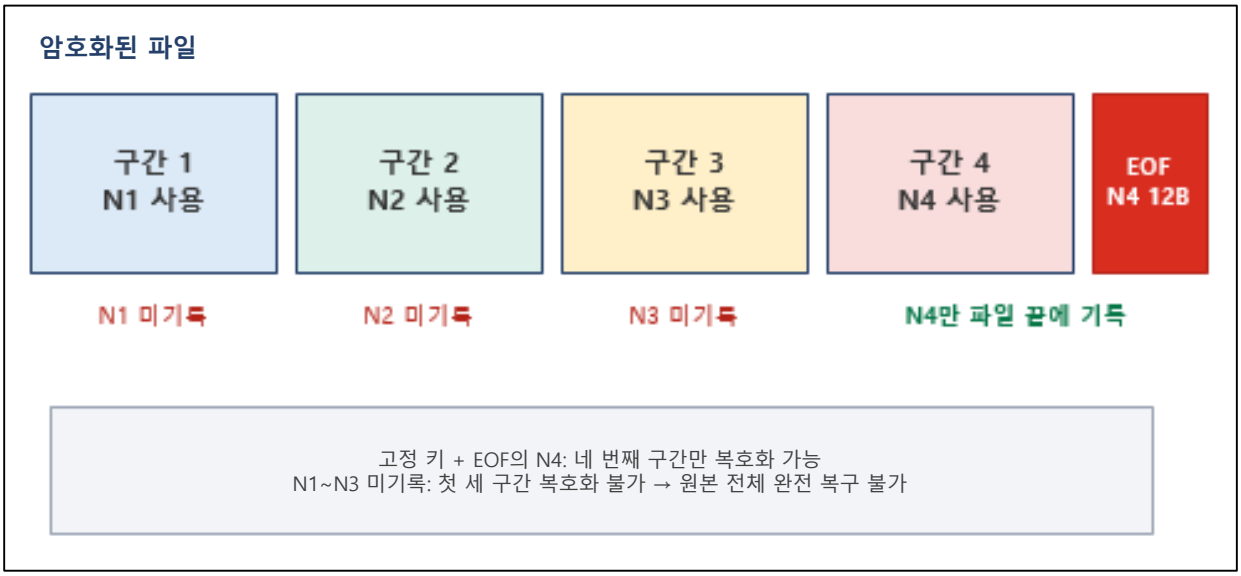


< 그림 26 > 네 구간에 각각 적용되는 12바이트 난스 N1~N4

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

파일 끝에 N4만 저장되는 구조와 복구 영향

암호화 함수는 N1부터 N4까지 만들면서 같은 12바이트 공간을 계속 덮어쓴다. 난스를 담아 둘 자리가 하나뿐이라, 네 구간을 모두 처리하고 나면 그 자리에는 N4만 남는다. 그래서 고정 키와 N4로 되돌릴 수 있는 범위는 네 번째 구간뿐이다. 앞의 세 구간을 복호화하려면 N1~N3가 필요한데 파일 어디에도 남지 않아, 원본 상태로 복구하려면 이 값들이 덮어써지기 전에 메모리에서 확보했거나 별도 원본 백업이 있어야 한다.



< 그림 27 > 대용량 파일 끝에 N4만 남는 저장 구조

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

암호화 전 파일명 변경과 크기별 처리

암호화보다 이름 변경이 먼저 일어나며, <원본>.vect로 바꾸는 데 성공하면 그 파일을 열어 크기에 맞는 방식을 고른다. 0바이트 파일은 이름만 바뀌고 끝난다.

파일 크기	암호화 방식	추가 특징 및 복구 영향
공통	<원본>.vect로 이름을 변경한 뒤 암호화를 시작	이름 변경에 성공한 파일만 다음 단계로 넘어간다.
128 KiB 이하	32 KiB 버퍼에 파일 전체를 한 번 읽어 암호화	파일 끝에 12바이트 난스 하나를 기록하며, 고정 키와 해당 난스로 복호화할 수 있다. 32 KiB를 넘는 파일은 읽기 길이가 버퍼 크기를 넘어선다.
128 KiB 초과	0, N/4, 2N/4, 3N/4 위치에서 각각 최대 32 KiB 암호화	네 구간이 모두 처리돼도 N4만 파일 끝에 기록된다. 고정 키와 N4로는 네 번째 구간만 복호화할 수 있으며, 원본과 동일한 복구에는 N1~N3 또는 백업이 필요하다.

< 표 3 > 파일 크기별 암호화 방식과 복구 영향

```
else
{
  v8 = lpFileName;
}
v8[v7] = 0;
if ( MoveFileExW(a2, v8, 1u) )                // MoveFileExW로 원본을 .vect 이름으로 먼저 바꾼다.
```

< 그림 28 > 암호화 전에 .vect로 변경되는 파일명

```
      SetFilePointer(v13, 0, 0LL, 2u);        // 변환 뒤 파일 포인터를 EOF로 이동한다.
      WriteFile(v13, Buffer, 0xCu, &NumberOfBytesWritten, 0LL);// 마지막 12바이트 난스 하나만 EOF에 기록한다.
      _InterlockedAdd64((a1 + 8), FileSize.QuadPart);// 원본 파일 전체 크기를 처리 통계에 더한다.
    }
    CloseHandle(v13);                         // 크기 조회·읽기·변환·쓰기 결과와 무관하게 유효하게 열린 파일 핸들을 닫는다.
  }
}
if ( Block != &v27 )                          // 확장자 문자열이 힙 버퍼를 사용했으면 해제한다.
  j_j_free_4(Block);
```

< 그림 29 > 파일 끝에 마지막 12바이트 난스 기록

랜섬노트 생성 위치와 기록 내용

경로 탐색 스레드는 탐색한 각 디렉터리와 바탕화면에 !!!_READ_ME_!!!.txt를 만든다. 같은 이름이 이미 있으면 손대지 않는다. 처음 만들 때는 암호화 시도 건수와 원본 크기 합계를 모두 0으로 기록하고, 각 시작 경로의 탐색이 끝나면 누적값으로 갱신한다.

랜섬노트 경로를 만들 때는 시작 경로 문자열 끝에 백슬래시를 붙이지 않고 파일명을 그대로 이어 붙인다. 시작 경로가 D:₩data라면 생성 경로는 D:₩data!!!_READ_ME_!!!.txt가 된다.

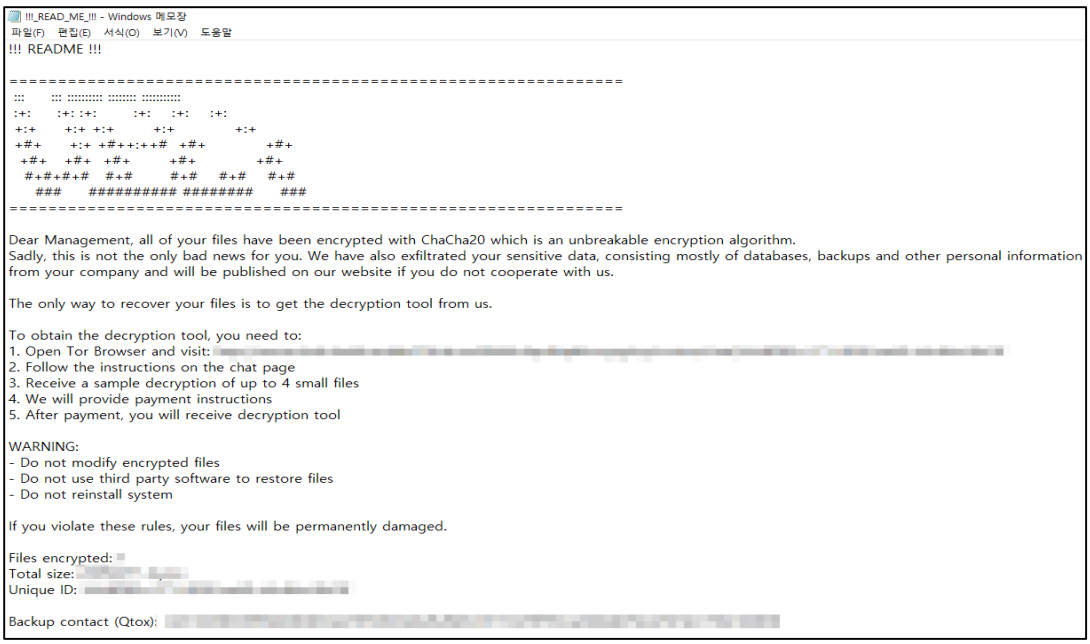
노트 본문에는 Tor 주소와 qTox 연락처, 감염 시스템마다 다른 피해자 ID가 들어간다. 여기에 자료를 이미 확보했다는 주장도 함께 적혀 있다.

```
v145 = strlen(v143);
StdOstream_InsertCharsPadded(v284, v143, v145);
v146 = StdOstream_InsertCharsPadded(v284, *victimId, victimId[1]);// victimId를 랜섬노트의 Unique ID로 추가한다.
BYTE2(Block[0]) = 73;
LOWORD(Block[0]) = 2001;
v147 = GetOrRegisterLazyGlobalSlot(qword_1401213C0);// Unique ID 뒤에 빈 줄 두 개를 추가한다.
v148 = GetOrRegisterLazyGlobalSlot(qword_140122F80);
```

< 그림 30 > 랜섬노트에 피해자 ID 삽입

```
BuildRansomNoteAndAttemptWriteCopies(&v3, targetRoot, *ctx, ctx[1], ctx + 33, ctx + 37);// 루트 경로와 바탕화면에 랜섬노트 기록을 시도한다.
if ( sub_14007ED40(targetRoot, "C:", 0LL, 2uLL) != -1 || sub_14007ED40(targetRoot, "c:", 0LL, 2uLL) != -1 )// targetRoot에 C: 또는 c:가 포함됐는지 검사한다.
{
  AttemptCreateRansomWallpaperBMP_ToTemp(Block);// TEMP에 1920×1080 BMP 생성을 시도한다.
  if ( Block[1] )                               // BMP 경로가 있으면 바탕화면 적용을 시도한다.
    AttemptApplyWallpaper_SPI(Block);           // SystemParametersInfoW로 바탕화면 변경을 요청한다.
  if ( Block[0] != &v6 )
    j_j_free_4(Block[0]);
}
if ( v3 != &v4 )                                // 생성한 랜섬노트 본문 문자열을 해제한다.
  j_j_free_4(v3);
```

< 그림 31 > 경로 탐색 완료 후 랜섬노트 통계 갱신



< 그림 32 > 생성된 랜섬노트 실제 출력 화면

랜섬웨어 바탕화면 이미지 생성 및 설정

1920×1080 크기의 24비트 BMP를 %TEMP%₩dvm3_wall.bmp에 만들고 VECT 2.0과 YOUR FILES HAVE BEEN ENCRYPTED, 피해자 ID를 그려 넣는다. 단, 암호화를 시작한 경로에 C: 또는 c:가 들어 있어야 이미지 생성과 바탕화면 변경으로 넘어간다.

이때 경로 전체를 따지지 않고 해당 문자가 들어 있는지만 확인하므로, 경로 중간에 있어도 조건을 통과한다. 조건을 통과하면 BMP를 만든 직후 Windows 바탕화면 변경 함수를 호출한다.

```
if ( 0x3FFFFFFFFFFFFFFLL - *v47 < v5 )        // 동적 표시 문자열에 접두사 "ID: "를 붙인다.
  sub_14011C480("basic_string::append");
std_string_append_cstr(&v46, *(&qword_14011F6E0 + 25), v5);// 전역 설정 객체의 피해자 ID 바이트를 길이 값만큼 그대로 덧붙인다.
GetTextExtentPoint32A(CompatibleDC, v46, v47[0], &psizl);
TextOutA(CompatibleDC, (1920 - psizl.cx) / 2, 860, v46, v47[0]);// "ID: "와 피해자 ID를 합친 문자열을 x축 중앙, y=860에 출력한다.
memset(&bmi.bmiHeader.biCompression, 0, 28);
*&bmi.bmiHeader.biSize = 0x7800000028LL;
*&bmi.bmiHeader.biHeight = 0x18000100000438LL;
v6 = sub_140119E10(0x5EEC00uLL);              // 1920*1080*3 = 6,220,800바이트의 픽셀 버퍼를 할당하고 0으로 초기화한다.
*v6 = 0;
v7 = v6;
memset(v6 + 1, 0, 0x5EEBFFuLL);
GetDIBits(CompatibleDC, hbm, 0, 0x438u, v7, &bmi, 0);// 1920×1080·24bpp BITMAPINFOHEADER로 픽셀을 가져온다.
DeleteObject(ho);                             // 이전에 선택된 GDI 객체를 복원하지 않은 채 글꼴 3개와 비트맵을 삭제하고 결과를 무시한다.
DeleteObject(v37);
DeleteObject(v3);
DeleteObject(hbm);
DeleteDC(CompatibleDC);
ReleaseDC(0LL, hDC);
v52[1] = 0x18000100000438LL;
v41 = 0;
v52[2] = 0x5EEC0000000000LL;
v52[0] = 0x7800000028LL;                      // 1920×1080, 24bpp BMP 헤더를 구성한다.
v52[3] = 0xB1300000B13LL;
v52[4] = 0LL;
v39 = 0x5EEC364D42LL;
v40 = 3538944;
GetTempPathA(0x104u, Buffer);                 // 260바이트 버퍼에 GetTempPathA를 호출한다.
Block[0] = 0x95C4142C365EFDF1LL;
LODWORD(Block[1]) = 1750181369;
WORD2(Block[1]) = 25347;
v8 = GetOrRegisterLazyGlobalSlot(qword_140120EC0);// 14바이트 정적 블록을 파일명 복호화용 버퍼에 준비한다.
```

< 그림 33 > 1920×1080 24비트 BMP 생성 구조

```
BuildRansomNoteAndAttemptWriteCopies(&v3, targetRoot, *ctx, ctx[1], ctx + 33, ctx + 37);// 루트 경로와 바탕화면에 랜섬노트 기록을 시도한다.
if ( sub_14007ED40(targetRoot, "C:", 0LL, 2uLL) != -1 || sub_14007ED40(targetRoot, "c:", 0LL, 2uLL) != -1 )// targetRoot에 C: 또는 c:가 포함됐는지 검사한다.
{
  AttemptCreateRansomWallpaperBMP_ToTemp(Block);// TEMP에 1920×1080 BMP 생성을 시도한다.
  if ( Block[1] )                             // BMP 경로가 있으면 바탕화면 적용을 시도한다.
    AttemptApplyWallpaper_SPI(Block);         // SystemParametersInfoW로 바탕화면 변경을 요청한다.
  if ( Block[0] != &v6 )
    j_j_free_4(Block[0]);
}
if ( v3 != &v4 )                              // 생성한 랜섬노트 본문 문자열을 해제한다.
  j_j_free_4(v3);
```

< 그림 34 > BMP 생성 후 바탕화면 설정 호출

< 그림 35 > 통제 환경에서 확인한 VECT 2.0 바탕화면 이미지

상세 출력(-v)과 통계의 의미

상세 출력 옵션(-v)을 주면 실행 중 진행 상황을 콘솔에 출력한다. 약 500ms 간격으로 암호화 시도 건수를 표시하고, 로컬 암호화가 끝난 뒤 표시 파일 기록과 --gpo 원격 실행 사이에 최종 통계를 한 번 출력한다.

최종 통계에는 Files encrypted와 Data encrypted 두 값이 나온다. 앞의 값은 암호화 함수를 호출한 직후 증가하므로 이름 변경이나 파일 열기에 실패한 파일, 이름만 바뀐 0바이트 파일도 함께 집계된다. 뒤의 값은 대상 파일의 원본 크기를 그대로 더한 값이라, 128 KiB를 넘는 파일이 실제로는 최대 128 KiB만 바뀌어도 전체 크기가 합계에 반영된다.

```
while ( 1 )
{
  result = *(*(a1 + 8) + 56LL);                 // 실행 플래그가 0이 될 때까지 반복한다.
  if ( !result )
    break;
  VerboseProgress_PrintEncryptedAttemptCount(*(a1 + 8));// 암호화 시도 건수를 verbose 진행률로 출력한다.
  Sleep(0x1F4u);                                // 각 출력 뒤 500ms 대기한다.
}
return result;
```

< 그림 36 > 약 500ms 간격의 암호화 시도 건수 출력

--gpo 원격 실행

--gpo가 켜져 있으면 원격 실행을 담당하는 스레드 세 개를 만든다. 각 스레드는 Active Directory에서 조회한 컴퓨터의 C$₩ProgramData에 실행 파일 복사를 시도하고, 복사에 성공한 대상에서 서로 다른 방식으로 실행을 시도한다. 세 스레드는 각각 WMI로 프로세스를 생성하거나, DCOM 프로토콜을 사용하는 CIM으로 프로세스를 생성하거나, CIM 원격 연결 세션에서 SYSTEM 권한 예약 작업을 등록한다.

```
if ( v93 )                              // --gpo가 켜지면 원격 실행 스레드 3개를 준비한다.
{                                       // --gpo 모드 정보 출력
  if ( v98 )
  {
    BYTE6(Stream[1].size) = -71;
    Stream[1].data = 0x98DDE098DFB3309DLL;
    Stream[0] = _mm_loadu_si128(&xmmword_140125380);
    LODWORD(Stream[1].size) = -2081610541;
    WORD2(Stream[1].size) = -29741;
    v32 = GetOrRegisterLazyGlobalSlot(qword_140120940);
    v33 = GetOrRegisterLazyGlobalSlot(qword_140122500);
    v34 = v33;
    if ( !*v32 )
    {
      v35 = _mm_loadu_si128(Stream);
      v33[31] = 1;
      v36 = _mm_loadu_si128((&Stream[0].size + 7));
      *v33 = v35;
      *(v33 + 15) = v36;
      *v32 = 1;
      RegisterProcessExitCallback(nullsub_92, v33);
    }
    if ( v34[31] )
    {
      for ( n = 0LL; n != 31; ++n )
        v34[n] ^= 17418095960374068157uLL >> (8 * (n & 7u));
      v34[31] = 0;
    }
    Console_Printf_CLI(v34);            // verbose 모드에서 원격 실행 단계 시작 메시지를 Console_Printf_CLI로 출력한다.
  }
  Stream[0].data = v109;
  v37 = sub_14007D250(0LL, 0LL, 1uLL, "vector::_M_realloc_insert");// 자격증명 컨텍스트를 공유하는 원격 실행 스레드 3개를 준비한다.
  v38 = v37;
  if ( v37 )
  {
    v39 = sub_1400E97D0(v37);
    sub_1400F3C90(v39, Stream);         // WMI Win32_Process.Create 실행 스레드를 생성한다.
```

< 그림 37 > 원격 실행 스레드 생성

대용량 파일 완전 복구를 막는 난스 오류를 가진 랜섬웨어

실행 흔적 삭제와 선택적 자기 삭제

원격 실행 스레드가 끝나면 흔적을 지운다. 관리자 권한이면 Application과 Security, System, Windows PowerShell 이벤트 로그 삭제를 시도하고, PSReadLine 명령 기록과 Recent 폴더 항목은 권한과 무관하게 지운다. 다만 Recent는 하위 폴더까지 들어가지 않고 바로 아래 항목만 처리한다.

흔적을 지운 뒤, --stealth가 켜져 있으면 자기 삭제를 시도한다. 현재 실행 파일 경로를 삭제 명령에 넣고, ping으로 시간을 지연시킨 뒤 실행한다.

```
AttemptDeletePowerShellPSReadLineHistory();// PSReadLine 명령 기록 파일 삭제를 시도한다.
AttemptDeleteRecentFolderEntries_NonRecursive();// Windows Recent 항목 삭제를 시도한다.
```

< 그림 38 > PSReadLine과 Recent 사용 기록 삭제

```
if ( v97 )                              // --stealth가 켜지면 지연 자기 삭제를 시도한다.
{
  GetModuleFileNameA(0LL, Filename, 0x104u);// 현재 실행 파일 경로를 자기 삭제 대상으로 사용한다.
  std_string_from_cstr(&StartupInfo, Filename);// 현재 실행 파일 경로를 자기 삭제 명령에 추가한다.
  Stream[2].data = 0x5B16ED7BE6915DA3LL;
  Stream[0] = _mm_loadu_si128(&xmmword_1401253C0);
  LODWORD(Stream[2].size) = -1551609179;
  Stream[1] = _mm_loadu_si128(&xmmword_1401253D0);
  WORD2(Stream[2].size) = -13003;
  v48 = GetOrRegisterLazyGlobalSlot(qword_140120980);
  v49 = GetOrRegisterLazyGlobalSlot(qword_140122540);
  v50 = v49;
  if ( !*v48 )
  {
    v49[46] = 1;
    qmemcpy(v49, Stream, 0x2EuLL);
    *v48 = 1;
    RegisterProcessExitCallback(nullsub_128, v49);
  }
  if ( v50[46] )                        // 키 0x3D39CD1783F57D85로 46바이트를 복호한다.
  {
    for ( kk = 0LL; kk != 46; ++kk )
      v50[kk] ^= 0x3D39CD1783F57D85uLL >> (8 * (kk & 7u));
    *(v50 + 46) = 0;
  }
  std_string_from_cstr(&ProcessInformation, v50);// ping 지연 뒤 실행 파일을 삭제하는 명령을 구성한다.
```

< 그림 39 > --stealth 기반 지연 자기 삭제 명령

4) 결론

분석한 VECT 2.0은 관리자 권한에서 Defender 설정을 바꾸고 서비스와 프로세스를 종료한 뒤, 볼륨 섀도 복사본과 이벤트 로그를 지운다. 로컬과 네트워크 경로의 파일명을 .vect로 바꿔 암호화하며, WMI와 CIM을 이용한 원격 실행도 시도한다.

가장 큰 특징은 파일 크기에 따라 갈리는 암호화 처리다. 128 KiB를 넘는 파일에는 앞 세 구간을 되돌릴 N1~N3가 남지 않아, 이 값이나 원본 백업이 없으면 원본과 동일하게 복구할 수 없다. 128 KiB 이하 파일도 32 KiB 버퍼에 파일 전체를 읽어 들이므로, 32 KiB를 넘는 파일에서는 읽기 길이가 버퍼 크기를 넘어선다.

방어 관점에서는 개별 파일 해시나 확장자 변경 같은 단일 지표만으로는 부족하다. 권한 확인과 보안 설정 변경, 복구 수단 무력화, 공유 경로 탐색과 원격 실행, 대량 파일 암호화로 이어지는 행위 흐름을 연계해 탐지해야 한다. 관리 공유와 도메인 계정의 권한을 최소화하고, 운영망과 분리되거나 삭제와 변조가 제한된 백업 체계도 함께 마련해야 한다.

감염이 확인된 뒤에는 대응 순서가 중요하다. 앞서 말한 N1~N3는 실행 중 메모리에만 존재하므로, 프로세스를 종료하기 전에 사고 대응 절차에 따라 휘발성 메모리를 확보해야 한다.

5) IOCs

분석 대상 SHA-256

8ee4ec425bc0d8db050d13bbff98f483fff020050d49f40c5055ca2b9f6b1c4d

관련 VECT 2.0 파일 SHA-256

Windows | 9c745f95a09b37bc0486bf0f92aad4a3d5548a939c086b93d6235d34648e683f

Windows | e512d22d2bd989f35ebaccb63615434870dc0642b0f60e6d4bda0bb89adee27a

Linux | e1fc59c7ece6e9a7fb262fc8529e3c4905503a1ca44630f9724b2ccc518d0c06

ESXi | a7eadcf81dd6fda0dd6affefaffcb33b1d8f64ddec6e5a1772d028ef2a7da0f2

ESXi | 58e17dd61d4d55fa77c7f2dd28dd51875b0ce900c1e43b368b349e65f27d6fdd

6) YARA Rule

```
rule Ransomware_VECT_2_0
{
    meta:
        description = "VECT 2.0 ransomware"
        author = "Threat Analysis Center"
        sample_sha256 = "8ee4ec425bc0d8db050d13bbff98f483fff020050d49f40c5055ca2b9f6b1c4d"
        analysis = "binary and controlled-environment dynamic analysis"
        scope = "exact sample; related VECT 2.0 files untested"

    strings:
        $core_banner = "VECT 2.0" ascii
        $core_note = "!!!_READ_ME_!!!.txt" ascii
        $core_wall = "YOUR FILES HAVE BEEN ENCRYPTED" ascii
        $ad_query = "[adsisearcher]'objectCategory=computer'" ascii
        $wmi_exec = "Invoke-WmiMethod -ComputerName" ascii
        $cim_dcom = "New-CimSessionOption -Protocol Dcom" ascii
        $sched_task = "Register-ScheduledTask" ascii
        $force_safe = "--force-safemode" ascii
        $no_gpo = "--no-gpo" ascii
        $no_mount = "--no-mount" ascii
        $no_stealth = "--no-stealth" ascii
        $stats = "Files encrypted" ascii

    condition:
        filesize < 3MB and
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c)) == 0x00004550 and
        uint16(uint32(0x3c) + 0x18) == 0x020b and
        all of ($core*) and
        3 of ($ad_query, $wmi_exec, $cim_dcom, $sched_task,
              $force_safe, $no_gpo, $no_mount, $no_stealth, $stats)
}
```

7) 참고 자료

Check Point Research, "VECT: Ransomware by design, Wiper by accident," 2026-04-28.
https://research.checkpoint.com/2026/vect-ransomware-by-design-wiper-by-accident/