asec.ahnlab.com /ko/90671/

취약한 난수 생성 함수를 사용하는 Gunra 랜섬웨어 분석 (리눅스 환경 대상 유포, ELF 형태)

⋮ 10/21/2025

2025년 4월부터 활동을 시작한 Gunra 랜섬웨어 그룹은 최근 다양한 국가와 산업의 기업들을 대상으로 지속적인 공격을 감행하고 있으며, 국내에서도 피해 사례가 확인되고 있다. 유포에 확인된 Gunra 랜섬웨어는 윈도우 환경 대상의 EXE 파일 형식과, 리눅스 환경 대상의 ELF 파일 형식이 있다. 본 게시글에서는 리눅스 대상의 ELF 파일 형식의 Gunra 랜섬웨어의 주요 특징과 암호화 방식, 그리고 복호화가 가능한 기술적 이유를 분석하여, 향후 유사한 위협에 효과적으로 대응할 수 있도록 인사이트를 제공하고자 한다.

Gunra 랜섬웨어 그룹은 2025년 4월부터 활동을 시작했으며, 주로 Windows와 Linux 시스템을 표적으로 다양한 국가와 산업의 기업들을 대상으로 지속적인 공격을 수행하고 있다. 다른 랜섬웨어 그룹과 같이 감염된 시스템의 파일을 암호화하고 피해 기업의 민감 데이터를 탈취하며, 몸값이 지불되지 않을 경우 해당 정보를 공개한다.

분석 내용

Gunra 랜섬웨어는 실행 초기 단계에서 전달된 인자 값에 대한 유효성 검사를 수행하며, 필요한 모든 인자 값이 올바르게 전달된 경우에만 정상 실행된다.

인자 값	행위	필요 여부
-t, –threads	암호화에 사용할 스레드 수	필요
-p, –path	암호화 대상 경로	필요
-e, –exts	암호화 대상 파일 확장자	필요
-r, –ratio	암호화 비율 (MB 단위)	필요
-k, –keyfile	RSA 공개키 파일 경로	필요
-s, –store	ChaCha20 암호화 알고리즘 키 저장 경로	선택
-l, –limit	최대 암호화 크기 (GB 단위)	선택

[표 1] 인자 값 별 행위

암호화 준비

Gunra 랜섬웨어는 –path 인자로 전달받은 경로를 대상으로 암호화를 수행한다. 암호화 방식은 경로의 유형에 따라 크게 파일 암호화와 디스크 암호화로 나뉜다. 파일 암호화의 경우, 각 파일마다 독립적인 암호화 스레드를 생성하여 암호화를 진행하며, 생성되는 스레드의 수는 –threads 인자를 기반으로 최소 1개에서 최대 100개까지 설정할 수 있다.

암호화 대상 파일의 확장자는 –exts 인자를 통해 지정한다. “all”을 전달하면 해당 경로에 존재하는 모든 파일을 대상으로 암호화를 수행하며, 특정 확장자를 지정하는 경우에는 최대 32개까지 설정할 수 있다. 또한 전달된 경로의 유형에 따라 암호화 동작 방식이 달라지는데, 경로가 파일인 경우에는 확장자 검사 없이 바로 암호화 스레드가 생성되어 암호화가 수행된다. 단, –exts 옵션에 “encrt” 값이 설정된 경우에는 해당 파일에 대한 암호화를 수행하지 않는다

폴더 경로가 전달된 경우에는 하위 경로에 존재하는 모든 파일을 대상으로 암호화 제외 대상 확인 및 확장자 검사를 수행한 후, 지정된 확장자를 가진 파일에 한해 각각의 암호화 스레드를 생성하여 암호화를 수행한다. 디스크 경로가 전달된 경우에는 –exts 옵션에 “disk” 값이 설정되고 –store 인자를 통해 암호화 키 저장 경로가 지정된 경우에만 해당 디스크 전체를 대상으로 암호화를 수행한다.

암호화 제외 대상 파일 및 확장자
R3ADM3.txt, .encrt

[표 2] 암호화 제외 대상 파일 및 확장자

파일 및 디스크 암호화

파일 대상의 암호화는 –store 인자 값이 설정된 경우와 설정되지 않은 경우 모두에서 동작한다. 암호화 알고리즘은 ChaCha20을 사용하며, 암호화에 사용되는 32바이트 크기의 키와 12바이트 크기의 Nonce 값은 매번 새로 생성된다. –store 인자 값이 설정된 경우, ChaCha20 암호화 알고리즘에 사용된 키는 RSA 공개키로 한 번

암호화된 후, 해당 경로에 .keystore 확장자로 저장된다. 반면, –store 인자가 설정되지 않은 경우에는 암호화된 파일의 끝에 해당 키가 삽입된다.

```
chacha20_init_state(v11, a1, a2, a3);
for ( i = 0; ; i += 64 )
{
  result = i;
  if ( (int)i >= a6 )
    break;
  chacha20_block(v11, v10, 20LL);
  ++v12;
  for ( j = i; j <= (int)(i + 63) && j < a6; ++j )
    *(_BYTE *)(j + a5) = v10[j - i] ^ *(_BYTE *)(j + a4);
}
return result;
```

[그림 1] ChaCha20 암호화 알고리즘

[그림 2] –store 인자 값에 따른 암호화 키 저장 방식

파일 암호화는 인자로 전달받은 –limit 및 –ratio 인자 값에 따라 설정된 암호화 제한 및 비율에 기반하여 서로 다른 방식으로 진행된다. 암호화 방식은 먼저 1MB 크기를 암호화한 후, –ratio 인자 값이 설정되어 있는 경우, 해당 값만큼 MB 크기의 내용을 건너뛴 뒤, 다시 1MB 크기를 암호화하는 방식으로 반복된다.

또한, 암호화 대상 파일의 크기가 –limit 인자로 전달받은 값보다 클 경우에는 파일의 앞부분부터 해당 크기까지 암호화를 수행한다. 반면, 암호화 대상 파일의 크기가 –limit 인자로 전달받은 값보다 작을 경우에는 파일 전체를 암호화한다.

디스크 대상의 암호화는 –store 인자 값이 설정된 경우에만 동작한다. 암호화 알고리즘은 ChaCha20을 사용하며, 암호화에 사용되는 32바이트 크기의 키와 12바이트 크기의 Nonce 값은 매번 새로 생성된다. 암호화 방

식은 파일 암호화와 동일하게, 인자로 전달받은 –limit 및 –ratio 인자 값에 따라 설정된 암호화 제한 및 비율에 기반하여 서로 다른 방식으로 진행된다.

데이터 복호화

각 암호화 스레드는 ChaCha20 암호화 알고리즘을 사용하여 암호화를 진행한다. 이때 사용되는 32바이트 크기의 키와 12바이트 크기의 Nonce 값을 생성하는 함수에 암호학적 취약성이 존재하여, 보안성이 매우 낮은 난수가 생성되는 문제가 발견됐다.

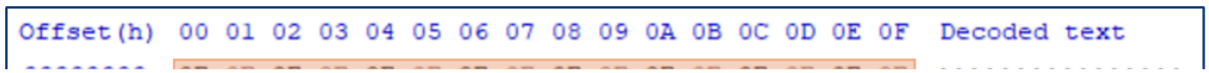
난수를 생성하는 함수는 time() 함수를 통해 초 단위의 현재 시간을 받아오고, 이를 기반으로 rand() 함수에 사용할 시드 값을 생성한다. 그러나 time() 함수의 반환 값을 바탕으로 시드 값을 생성하는 32회 및 12회의 반복문이 매우 짧은 시간 내에 실행되기 때문에 동일한 시드 값이 사용될 가능성이 높다. 이로 인해 rand() 함수가 동일한 바이트 값을 생성하게 되어, 결과적으로 동일한 바이트가 연속되는 32바이트 키와 12바이트 Nonce 배열이 생성된다. 즉, 암호학적으로 매우 취약한 키와 Nonce 값이 사용된다.

```
__int64 __fastcall generate_rand(__int64 a1_Buffer, unsigned int a2_Size)
{
  __int64 v2_CurrentTimeWithSeconds; // rdi
  __int64 result; // rax
  unsigned int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= a2_Size )
      break;
    v2_CurrentTimeWithSeconds = (unsigned int)time(0LL);// Get Current Time - Unit : Second
    srand(v2_CurrentTimeWithSeconds);           // Same Current Time (Second) >> Same Seed
    *(_BYTE *)(i + a1_Buffer) = rand();         // Same Seed >> Same Value
  }
  return result;                                // (Result) Buffer : Fill with Same Value
}
```

[그림 3] 암호화 키 및 Nonce 값 생성에 사용된 암호학적으로 취약한 함수

아래는 해당 취약한 난수 생성 함수로 생성된 ChaCha20 암호화 알고리즘의 키 및 Nonce 값이 포함된 배열이다. [그림 4]와 같이, 동일한 바이트가 연속되는 형태의 키와 Nonce 값이 생성된다.



[그림 4] 취약한 난수 생성 함수로 생성된 ChaCha20 키 및 Nonce 값 (RSA 암호화 전)

이러한 취약한 난수 생성 방식으로 인해 해당 버전의 Gunra 랜섬웨어로 암호화된 파일은 0x00부터 0xFF까지의 256가지 바이트 값을 기반으로 하는 Brute Force 기법을 통해 높은 확률로 복호화가 가능하다. 아래의 [그림 5]는 암호화된 파일을 대상으로 Brute Force 기법을 사용하여 암호화에 사용된 난수를 식별한 후, 이를 바탕으로 복호화를 수행한 예시이다.

[그림 5] 복호화 결과 (PDF 파일 시그니처 확인)

ELF 형태 vs EXE 형태

ELF 형태의 Gunra 랜섬웨어는 ChaCha20 암호화 알고리즘을 사용하며, 암호화에 필요한 키 값과 Nonce 값을 생성할 때 암호학적으로 취약한 난수 생성 함수를 사용한다. 이로 인해 높은 확률로 Brute Force 기법을 통해 암호화된 데이터를 복호화할 수 있다.

반면, EXE 형태의 Gunra 랜섬웨어는 ChaCha8 암호화 알고리즘을 사용하며, 키 값과 Nonce 값은 Cryptographic Service Provider(CSP)를 기반으로 한 CryptGenRandom() API를 사용해 생성한다. 해당 방식은 암호학적으로 안전한 난수 생성 방식이기 때문에 복호화는 사실상 불가능하다.

	ELF 형태	EXE 형태
암호화 알고리즘	ChaCha20	ChaCha8
난수 생성 방식	time() 함수 기반의 rand() 함수 사용	CryptGenRandom() API 사용
복호화 가능성	가능	불가능

[표 3] ELF 형태와 EXE 형태의 Gunra 랜섬웨어 특징 비교

안랩 대응 현황

안랩 제품군의 진단명과 엔진 날짜 정보는 다음과 같다.

[V3 진단]

- Ransomware/Win.Gunra.C5755872 (2025.05.20.03)
- Malware/MDP.Ransom.M1355 (2016.12.13.00)
- Ransom/MDP.Decoy.M1171 (2016.07.15.02)
- Ransom/MDP.Event.M1946 (2018.06.06.00)
- Ransom/MDP.Event.M1785 (2017.11.22.00)
- Ransom/MDP.Magniber.M1876 (2018.03.09.03)
- Ransom/MDP.Event.M4428 (2022.09.05.02)
- Ransom/MDP.Event.M4353 (2022.07.14.02)

[EDR 진단]

- SystemManipulation/EDR.Event.M2506 (2022.09.24.00)
- Ransom/EDR.Decoy.M2470 (2022.09.30.00)
- Ransom/MDP.Event.M1946 (2018.09.07.03)
- Malware/MDP.Ransom.M1876 (2018.09.07.03)
- Suspicious/MDP.Ransom.M12560 (2025.04.04.01)

MD5

7dd26568049fac1b87f676ecfaac9ba0

9a7c0adedc4c68760e49274700218507

추가 IoC는 ATIP에서 제공됩니다.