Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.32, NO.5, Oct. 2022 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2022.32.5.869

Deflate 압축 알고리즘에서 악성코드 주입 취약점 분석

김 정 훈†‡
바이너리랩 주식회사 (대표이사)

Malicious Code Injection Vulnerability Analysis in the Deflate Algorithm

Jung-hoon Kim†‡
BinaryLab (CEO)

요 약

본 연구를 통해 매우 대중적인 압축 알고리즘인 Deflate 알고리즘을 통해 생성되는 3가지 유형의 압축 데이터 블록 가운데 원본 데이터 없는 비 압축 블록(No-Payload Non-Compressed Block:NPNCB) 유형을 임의로 생성하여 정상적인 압축 블록 사이에 미리 설계된 공격 시나리오에 따라 삽입하는 방법을 통해 악의적 코드 또는 임의의 데이터를 은닉하는 취약점을 발견하였다. 비 압축 블록의 헤더에는 byte align을 위해서만 존재하는 데이터 영역이 존재하며, 본 연구에서는 이 영역을 DBA(Disposed Bit Area)라고 명명하였다. 이러한 DBA 영역에 다양한 악성 코드와 악의적 데이터를 숨길 수 있었으며, 실험을 통해 정상적인 압축 블록들 사이에 오염된 블록을 삽입했음에도 기존 상용 프로그램에서 정상적으로 경고 없이 압축 해제 되었고, 악의적 디코더로 해독하여 악성 코드를 실행할 수 있음을 보였다.

ABSTRACT

Through this study, we discovered that among three types of compressed data blocks generated through the Deflate algorithm, No-Payload Non-Compressed Block type (NPNCB) which has no literal data can be randomly generated and inserted between normal compressed blocks. In the header of the non-compressed block, there is a data area that exists only for byte alignment, and we called this area as DBA (Disposed Bit Area), where an attacker can hide various malicious codes and data. Finally we found the vulnerability that hides malicious codes or arbitrary data through inserting NPNCBs with infected DBA between normal compressed blocks according to a pre-designed attack scenario. Experiments show that even though contaminated NPNCB blocks were inserted between normal compressed blocks, commercial programs decoded normally contaminated zip file without any warning, and malicious code could be executed by the malicious decoder.

Keywords: Deflate, Zip, Non-compressed block, Steganography, Malware

I. 서 론

1.1 Deflate 압축 알고리즘의 압축 블록

Deflate 압축 알고리즘은 전 세계에서 가장 많이 사용되는 압축 포맷인 Zip 형식에서 채택한 핵심 압축 알고리즘이며, .apk, .gzip, .zlib, .pdf, .xlsb 등 수많은 파일 포맷에서 데이터 압축 방법에 적용된 알고리즘이다. Deflate 압축 알고리즘은 압축을 진행하면서 압축 대상 원본 데이터의 특성에 따라 압축 결과로서 3가지 종류의 압축 블록을 생성하면서 압축이 진행되며, 구체적으로는 Compressed block with dynamic huffman code(이하, Dynamic 블록), Compressed block with fixed huffman code(이하, Fixed 블록), Non-

Received(08. 18. 2022), Modified(09. 27. 2022), Accepted(10. 05. 2022)
† 주저자, powerzenith@naver.com
‡ 교신저자, powerzenith@naver.com(Corresponding author)

ompressed block(이하, 비 압축 블록)으로서 RFC 1951 표준으로 정의되어 있다[1]. Dynamic 블록은 원본 데이터로부터 높은 압축률을 가진 압축 결과 데이터를 생성할 수 있을 때 생성되며, Dynamic 블록보다는 압축률이 낮지만 압축하는 것이 더 이득일 경우에는 Fixed 블록이 생성되며, 압축이 되지 않거나 오히려 증가하는 경우에는 압축을 포기하고 원본 데이터와 원본 데이터를 길이를 저장한 비 압축 블록을 생성하게 된다.

1.2 비 압축 블록(Non-compressed block)의 구조와 DBA(Disposed Bit Area) 개념

비 압축 블록은 시작 위치에 따라 다양한 모습을 가질 수 있는데, byte 경계에서 시작하는 가장 간단한 예인 경우를 들자면, Fig. 1.과 같이 'byte 1'의 최하위 1비트(Least Significant Bit, 이하 LSB)는 RFC 1951 에서는 BFINAL이라고 부르며, 0 또는 1로서 현재 블록이 마지막 압축 데이터 블록 인지 여부를 나타내며, 참고로 1 인 경우 해당 압축 블록이 마지막 블록임을 나타낸다. 이후 MSB(Most Significant Bit) 방향으로 읽은 2비트는 블록의 type을 나타내며, 이러한 2비트를 BTYPE 이라고 하며 BTYPE 값을 MSB 방향에서 LSB 방향으로 읽었을 때 00 일 경우 비 압축 블록임을 나타내며, 01 일 경우 Fixed 블록, 10 일 경우 Dynamic 블록이며, 11은 예약된 값이다. 압축 블록(비 압축 블록포함)이 마지막 압축 블록인지 여부를 나타내는 BFINAL 1비트, 그리고 이후에 읽은 BTYPE 2비트가 블록의 종류를 나타내는 3비트는 Deflate 알고리즘이 생성하는 3가지 종류의 압축 블록이 가진 공통적 특징이다[1].

다시 Fig. 1.에서 'byte 1'을 살펴보면 BTYPE 영역으로부터 MSB 방향으로 5비트는 'byte 2'에서부터 2바이트(16비트)의 LEN 값의 시작되도록 byte 경계에 맞추는 byte align에 필요한 영역으로서, RFC 1951 표준에 따르면 이러한 영역의 데이터에 무엇을 채워야 하는지 대해서는 특별히 명시되어 있지 않고 무시된다고(ignored)만 되어 있으며[1], 특별히 의도치 않는 한 변수 초기화 과정에서 실무적으로 0 으로 채워지게 된다. 이 영역을 본 연구에서는 DBA(Disposed Bit Area)라고 명명하였다. 참고로 byte align은 바이트 중간에서 데이터가 끝날 경우 바이트 경계에서 데이터가 끝나도록

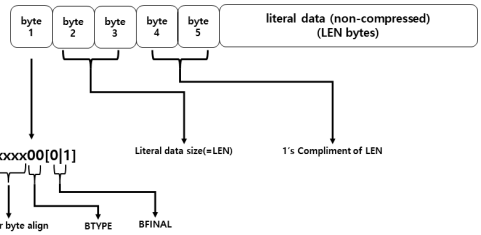

Fig. 1. General structure of non-compressed block in the Deflate algorithm

하는 일련의 절차를 나타내는 용어라는 의미로 본 논문에서 사용하였다. 한편 이후 'byte 2' 와 'byte 3'의 16비트 영역에 압축되지 않은 원본 데이터의 크기인 LEN을 표현하며,'byte 4' 와 'byte 5'의 16비트 영역에 LEN의 1의 보수를 표현하며, 이후 바이트부터 LEN byte 의 크기로 압축되지 않는 원본 데이터가 출력된다. 이러한 압축되지 못한 원본 데이터를 RFC 1951 에서는 literal data라고 한다[1].

한편, 실제 압축데이터에서는 DBA 시작 위치와 영역의 길이는 일정하지 않는데, 그 이유는 직전에 생성된 압축 블록이 끝나는 위치에 따라 byte align을 위해 필요한 비트 수가 달라지기 때문이다. Fig. 2.에서와 같이, 직전 압축 데이터 블록이 바이트의 중간에서 끝나면, 해당 바이트의 MSB 방향으로 직전 블록의 마지막 비트에 이어서 BFINAL 및 BTYPE을 나타내는 3비트로 구성된 prefix가 encoder로 부터 생성되고, byte align을 위해 MSB 방향으로 해당 바이트의 경계까지 7 비트가 DBA로써 할당된다. Fig. 2.에서 점선 화살표 및 숫자는 생성된 압축 데이터를 디코더가 물리적으로 읽는 순서이다. Deflate 알고리즘에서는 기본적으로 디코더가 데이터를 물리적으로 읽을 때 LSB → MSB 방향으로 읽으며, 읽어온 데이터를 MSB → LSB 또는 LSB → MSB 방향 중 어떤 방향으로 해석 할지에 대해서는 읽어온 데이터가 허프만 코드인지 여부 또는 압축 해제 시 동적으로 사전을 생성하기 위한 부가 정보 영역 등인지의 구체적으로 정보의 종류에 따라 RFC 1951에 미리 정해져 있다[1].

Dynamic 또는 Fixed 압축 블록의 경우 실제 압축된 코드에 대해 디코더는 LSB → MSB 방향으로 물리적으로 읽은 다음, 이 값을 해석할 때는 MSB → LSB 방향으로 읽어서 허프만 코드를 인식한 뒤, Dynamic 블록일 경우 동적으로 생성한 허프만 코드 트리, Fixed 블록일 경우 정적 허프만

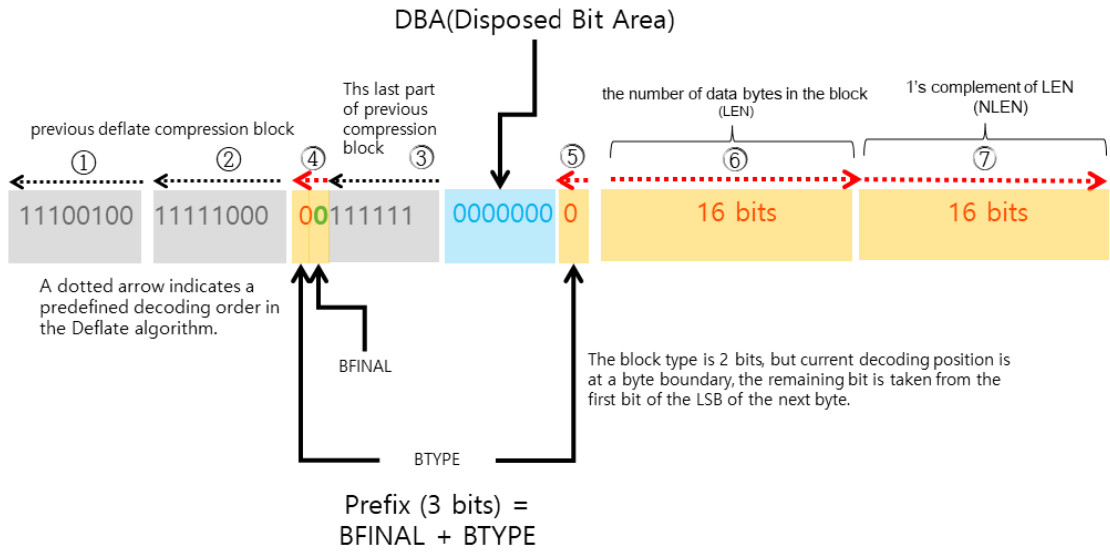

Fig. 2. Variation of DBA size in the non-compressed block

코드 트리를 이용하여 생성한 사전(Dictionary) 정보로부터 압축 해제 과정을 거친다[1]. 특히 Fig. 2에서 직전 압축 블록과 비 압축 블록의 경계가 존재하는 세 번째 바이트('00111111')를 살펴보면, 비 압축 블록을 구성하는 3비트의 prefix를 MSB 방향으로 읽어가면서 의미를 해독하게 된다. BFINAL에 해당하는 비트(0)은 마지막 압축 블록이 아니라는 의미이며, 이어서 세 번째 바이트의 MSB상의 마지막 1비트와 네 번째 바이트의 LSB상의 첫 1비트를 결합한 '00' 은 BTYPE 으로서 이 블록이 비 압축 블록임을 나타내며, 이어서 네 번째 바이트의 7개의 0 ('0000000') 은 byte align을 위한 DBA이다. Fig. 1.에서는 DBA가 5비트 크기였으나, Fig.2. 에서의 DBA 는 7비트 크기로서 비 압축 블록이 시작하는 위치에 따라 DBA 영역이 가변적임을 확인할 수 있다. 다음으로 다섯 번째 및 여섯 번째 바이트를 구성하는 16비트가 LEN을 나타내며, 일곱 번째 및 여덟 번째 바이트는 MSB에서 LSB 방향으로 16비트가 LEN의 1의 보수인 NLEN을 나타낸다.

그림에서는 보이진 않았지만, NLEN 영역 다음에 LEN bytes만큼의 압축되지 못한 원본 데이터인 literal data가 위치하게 된다.

한편, DBA 가 존재하지 않는 경우도 있는데, 아래 Fig. 3.과 같이 3비트의 prefix 영역이 byte 경계에서 우연히 끝나는 경우에는 byte align이 이미 달성된 상태이므로 비 압축 데이터의 크기를 나타내는 4 바이트의 영역(LEN 및 NLEN)이 다섯 번째 바이트의 byte boundary에서 자동적으로 시작되어 byte align을 위한 DBA 영역이 필요 없게 된다.

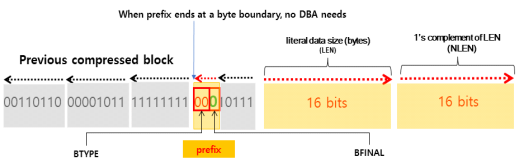

Fig. 3. The case where the DBA dose not exist because the prefix ends at the byte boundary

1.3 원본 데이터 없는 비 압축 블록(No-Payload Non-Compressed Block) 개요 및 활용

1.2절에서 언급한 비 압축 블록 가운데 특수한 형태로서, literal data라고 하는 원본 데이터의 그대로인 압축되지 못한 데이터를 담지 않는 형태의 블록이 있다. 이와 같은 블록을 본 연구에서는 원본 데이터 없는 비 압축 블록(No-Payload Non-Compressed Block; 이하, NPNCB)라고 하였다. NPNCB 또한 비 압축 블록의 한 유형이므로 블록이 시작되는 위치에 따라 DBA가 가변적이기 때문에 여러 형태를 가질 수 있으나, byte 경계에서 블록이 시작하는 간단한 사례의 경우로 표현하면, Fig. 4.와 같은 구

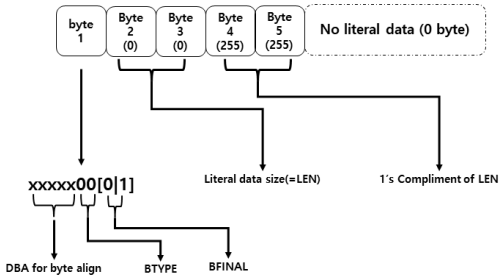

Fig. 4. The structure of No-Payload Non-Compressed Block (NPNCB)

조를 가지고 있다. literal data 가 존재하지 않으므로 LEN 이 0 이고, 따라서 LEN 의 16비트를 나타내는 byte 2와 byte 3 이 각각 0 이고 byte 4 와 byte 5 가 그 1의 보수인 255, 255로 채워진 블록이다. NPNCB의 DBA 영역도 1.2절에서 언급한 대로 직전 압축 데이터 블록이 끝나는 위치에 따라 byte align을 위해 0~7비트 길이로 존재할 수 있다.

NPNCB가 구체적으로 사용되고 있는 예로서, 데이터 압축 알고리즘으로 Deflate가 채택된 PPP Deflate Protocol 표준, TLS(Transport Layer Security), SSH(The Secure Shell) 표준이 있다[2,3,4]. 이러한 통신 프로토콜에서는 연속적인 데이터 스트림(data stream)을 패킷(packet) 단위로 나누어 처리하게 된다. 그런데 1.2절에서 언급했듯이 Deflate의 블록은 기본적으로 bit 단위로 압축 데이터를 생성하므로 반드시 바이트 경계에서 끝나지는 않는다. 이렇게 바이트의 중간에서 압축 데이터가 종료될 경우, 바이트 단위의 패킷으로 처리되는 입출력 특성상 바이트의 경계에서 압축 데이터가 종료되도록 하여야 할 필요성이 있으며, 이 때 NPNCB가 사용될 수 있다. 구체적으로 Fig. 5.를 보면, 세 번째 바이트('01111111')에서, LSB → MSB 방향으로 6 비트('111111') 까지가 직전 압축 블록의 압축 데이터라고 했을 때, 직전 압축 블록이 바이트 중간에서 끝나므로, NPNCB 를 압축 데이터에 이어서 마지막 압축 블록(BFINAL=1)으로서 연결하면 바이트의 경계에서 데이터가 끝나게 된다. 또는 Zlib 와 같은 압축 파일 포맷이 사용되는 SSH transport layer 프로토콜에서는 현재의 압축 블록의 생성이 끝났고 압축 데이터를 출력(전송)할 것이라는 'Partial flush'라고 부르는 작동을 위해 NPNCB를 마지막 블록으로 부가하기도 하며[4], buffer 영역에 저장되어 있는 압축 데이터를

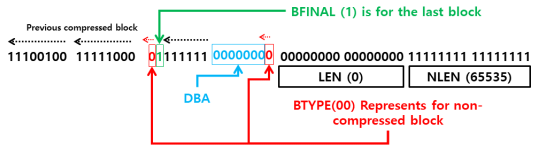

Fig. 5. Consecutive compressed data blocks dividing with NPNCB in the PPP, TLS, SSH

I/O 장치 등으로 쓰기 위한 'Full flush'라는 용도로도 사용된다. 즉 NPNCB를 만나면 decoder는 버퍼에 저장되어 있던 압축 해제 데이터를 I/O장치에 writing하고 압축 결과 저장용 임시 버퍼를 비운 뒤, 새롭게 다음 압축 데이터를 읽어서 압축 해제를 계속 진행하는데 사용한다[5].

1.4 원본 데이터 없는 비 압축 블록(No-Payload Non-Compressed Block) 의 보안 취약점

1.3절에서 소개한 NPNCB는 주로 통신 프로토콜에서 압축 알고리즘으로서 Deflate를 사용할 경우 패킷 단위 처리를 위해 강제적으로 압축 데이터를 분할하여 전송하거나 압축 해제 과정에서 디코더에게 꽉 찬 출력 버퍼를 flush하기 위한 decoding 지시의 용도로 종종 활용되고 있다. 그런데 이 외의 일반적인 저장 장치에서 압축 파일 생성 및 해제 시에 NPNCB의 사용은 특별히 제한받지 않는 것으로 확인되었는데, 본 연구에서는 대표적 사례로서 매우 대중적인 압축 파일 형식인 Zip 형식을 기준으로 테스트 하였고, 이에 따르면 악의적 encoder를 이용하면 압축 블록 사이에 공격자가 의도한 대로 자유롭게 NPNCB를 삽입 할 수 있다. 또한 압축 데이터 생성 시 뿐만 아니라, 정상적으로 이미 다른 프로그램으로 압축된 압축 데이터 내의 압축 블록들의 경계를 인식하여 그 사이에 NPNCB를 생성하여 삽입할 수 있다.

NPNCB가 기존 압축 데이터 블록 사이에 의도적으로 삽입될 경우에는 현존하는 일반적인 압축 프로그램의 CRC(Cyclic Redundancy Check) 검사로는 NPNCB 로부터 압축 해제된 데이터 자체가 없으므로 압축 해제된 데이터를 이용하여 계산하는 CRC 값에 영향을 미치지 않기 때문에, NPNCB의 삽입으로 CRC 값 이상에 따른 경고나 에러가 발생하지 않고 정상적으로 압축 해제되므로, 사용자는 NPNCB 블록이 삽입되었다는 사실조차 인지하지 못하게 된다. 부연하자면 압축 프로그램의 CRC체크

는 압축 해제된 데이터를 기준으로 하는 것이며, 압축된 결과 데이터(즉 연속된 압축 블록들)의 바이너리를 기준으로 하는 것이 아니기 때문에 NPNCB가 추가되어 압축 결과 데이터가 자체가 변조된다고 하여도 CRC체크에서는 특별한 경고가 발생하지 않는다는 점이다. 이와 같은 CRC 체크 취약점은 추후 보완될 필요성이 있는 것으로 여겨진다.

또한 NPNCB의 DBA영역에 악성 데이터 또는 코드를 삽입할 경우에도 현재의 상용 압축 프로그램에서는 어떠한 경고가 발생치 않았다. 본 연구에서는 위와 같은 취약점을 사용한 악성코드 주입 및 악의적 데이터의 숨김이 가능하며, 이를 통해 악의적 코드 실행 가능성을 확인하였다.

한편, 저장 매체에 압축 파일을 보관하는 Zip 형식뿐만 아니라, 1.3절에서 언급한 통신 프로토콜 등에서 악의적 NPNCB를 추가로 더 삽입할 경우에도 버퍼를 비우는 동작 등이 여러 번 발생할 뿐이지 NPNCB의 삽입으로 압축 데이터의 해독에 영향을 주거나 치명적인 통신 오류가 발생할 가능성은 없는 것으로 여겨지며, 향후 통신 프로토콜 또는 Deflate를 쓰는 다른 압축 포맷 상에서의 NPNCB삽입에 따른 동태는 추가 연구할 필요가 있다.

II. 본 론

2.1 NPNCB 삽입 공격 개요

서론에서 제언한 바와 같이, NPNCB는 Deflate 알고리즘으로 생성되는 3가지 종류의 데이터 압축 블록들 사이에 Fig. 6.과 같이 임의로 생성하여 삽입 될 수 있다. 2.2절에서 언급할 예정이지만 한 가지 주목할 부분은 정상적으로 생성되는 모든 유형의 압축 블록 사이에 임의로 생성 및 삽입이 가능하다는 것이며, 반드시 정상 압축 데이터에 비 압축 블록(Non-compressed block)이 존재하여야만 NPNCB가 의존적으로 삽입 가능한 것은 아니다. 기존 압축 블록과 독립적으로 악의적 encoder가 새롭게 임의로 생성시킨 대등한 압축 블록이기 때문이다. 이렇게 정상 블록들 사이에 삽입하였음에도 decoding 결과 자체에 영향을 주지 않는다.

예를 들어 Fig. 6.을 다시 보면, 이미 다른 압축 프로그램으로 정상적으로 압축된 데이터가 5개의 압축 블록으로 구성되어 있다면, NPNCB는 주변 압축 블록이 어떤 종류의 것이라도(심지어 Dynamic 또는 Fixed 블록만 존재하더라도) 두 개의 블록 사이에 이어서 임의로 생성하여 삽입이 가능하며, 여러 개가 연속적으로 삽입될 수도 있다. 따라서 매우 다양한 삽입 공격 시나리오가 도출될 수 있다. 한편 이렇게 NPNCB의 임의 삽입 하는 경우에, 기존 압축 데이터 자체가 커지는 부분에 대응하여 Zip format의 헤더 영역에 존재하는 해당 압축 데이터의 compressed size 정보에 대해 크기 변화를 고려하여 수정을 추가로 해주는 부분은 필요하나 크게 어려운 과정은 아니며, Fig. 7.처럼 악성 encoder가 압축 파일을 생성하는 "동시"에 NPNCB를 삽입하는 경우라면 이미 NPNCB가 삽입된 크기만큼 압축 데

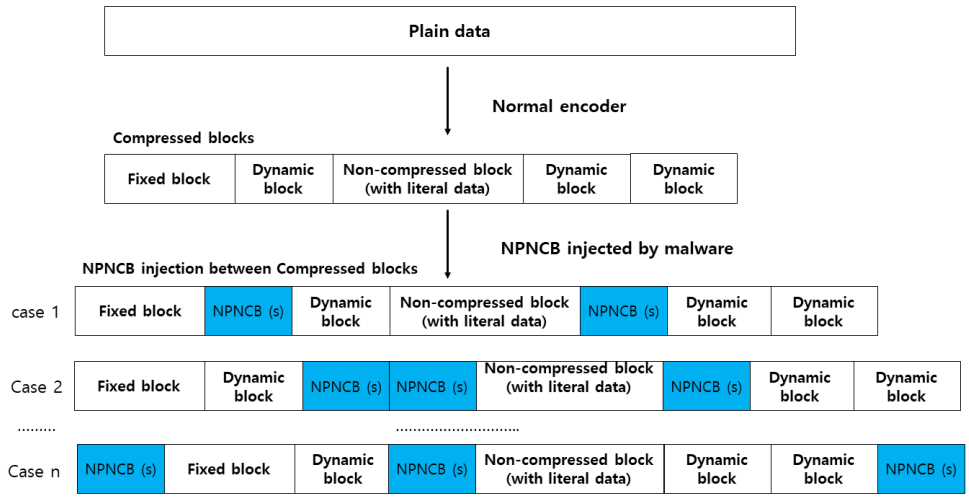

Fig. 6. The variability of NPNCB injection attack scenario design

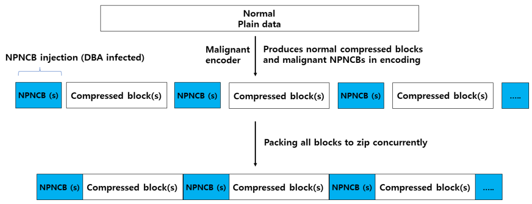

Fig. 7. The process of injecting malignant NPNCBs while initially encoding by malignant encoder

이터 사이즈도 이미 함께 계산되므로, 이러한 경우에는 Zip 포맷의 헤더 영역의 compressed size 데이터를 추가로 수정 과정은 필요하지 않을 것이다[6].

2.2 NPNCB 삽입 수행 및 압축 해제

2.1절에서 언급한 NPNCB를 실제로 본 연구에서 개발한 악의적 encoder로 Fig. 8.과 같이 두 가지의 예시 파일(m4a,xlsx)을 압축하면서 생성한 정상적인 압축 블록 사이에 모두 NPNCB를 삽입하였다.

Fig. 8.의 테스트 프로그램의 하단 텍스트 박스에 표시된 'DXDX...'와 같은 압축 블록 타입 결과를 해석하기 위한 범례로서 'D'는 Dynamic 블록을 타나내며, 'X'는 NPNCB 이며, 'N'은 비 압축 블록(Non-compressed block)을 나타낸다. 압축하기 위한 원본 데이터의 특성 및 Deflate 압축 변수 설정(압축률 최대로 설정)으로 인해 테스트 사례에서는 Fixed block이 생성되지 않았다. NPNCB를 삽입하지 않을 때에는 Table 1.과 같은 유형으로 전체 174개의 압축 블록이 순서대로 생성되었다.

그런데 NPNCB ('X' 표시된 블록)를 삽입한 결과로 Table 2.와 같이 348개의 압축 블록이 순서대로 생성되었다.

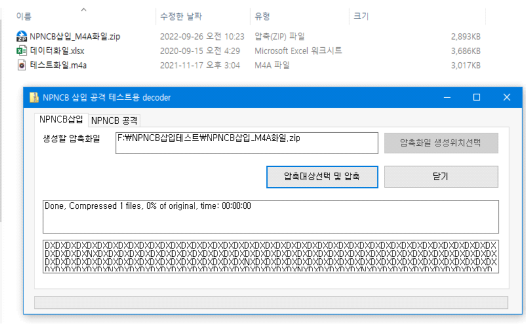

Fig. 8. The process of injecting NPNCBs to normal compressed blocks while the M4A file being encoded by test malignant encoder

Table 1. The compressed block type of m4a file by normal encoder

DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDNDD DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD DDDDNDDDDDDDDDDDDDDDDDDDDDDDDDDDDDNDDDDDDDDDDDDDDD DDNDDDDDNDDDDDDDDDDDDDDD

Table 2. NPNCBs injected compressed blocks by malignant encoder with m4a file

DXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD XDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXNXDXDXDX DXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD XDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDX DXDXDXNXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD XDXDXDXDXDXDXDXNXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDX NXDXDXDXDXNXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD

로 생성되었다.

이렇게 NPNCB가 삽입되었음에도 Fig. 9와 같이 정상적으로 압축 해제되었음을 알 수 있었다.

두 번째 NPNCB 삽입 사례로서, Fig. 10.과 같이 동일한 테스트용 엔코더로 엑셀 파일(.xlsx)을

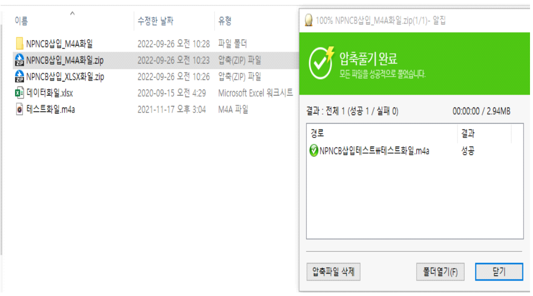

Fig. 9. NPNCBs inserted compressed file (.m4a) is normally decompressed by commercial encoder without any alerts

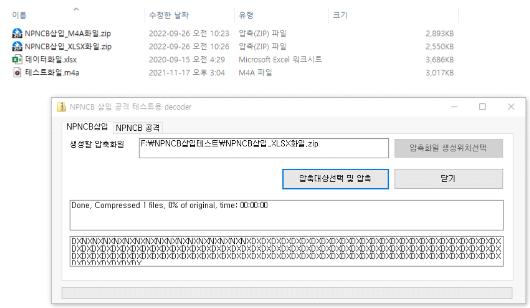

Fig. 10. The process of injecting NPNCBs to xlsx file compressed blocks while xlsx file being encoded by test malignant encoder

압축하였다.

정상적으로는 Table 3.과 같이 135개의 압축 블록이 순서대로 생성되었다.

그런데, NPNCB를 각 압축 블록 사이에 삽입하였을 경우 Table 4.와 같이 270개의 압축 블록이 순서대로 생성되었다.

이렇게 NPNCB가 압축 블록 사이에 모두 삽입되었을 경우에도 첫 번째 사례와 같이 Fig. 11.을 보면 압축이 정상적으로 해제되었음을 알 수 있다. 또한 주입한 NPNCB의 DBA영역을 임의의 데이터로 오염시킨 후 주입한 결과 역시 동일하게 정상적으로 압축해제 되었다.

Table 3. The compressed block type of xlsx file by normal encoder

DNNNNNNNNNNNNNNNNDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD

Table 4. NPNCBs injected compressed blocks by malignant encoder with .xlsx file

DXNXNXNXNXNXNXNXNXNXNXNXNXNXNXNXNXDXDXDXDXDXDXDXDXDX
DXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD
XDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDX
DXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXD
XDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDXDX
DXDXDXDXDXDXDX

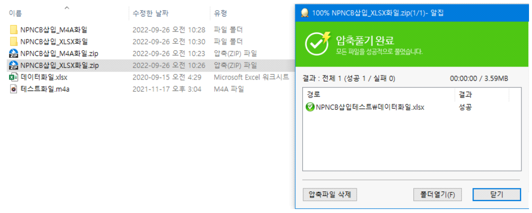

Fig. 11. NPNCBs inserted compressed file (.xlsx) is normally decompressed by commercial encoder without any alerts

2.3 NPNCB 삽입된 압축 데이터 decoding

2.2절에서 살펴 본 바와 같이, Deflate 알고리즘 자체만으로 본다면 NPNCB 삽입만으로 압축의 해제에 어떠한 영향을 주지 않으며, 현재 상용 프로그램에서는 Fig. 12.와 같은 이유로 보안 경고나 에러를 일으키지 않고 NPNCB 에 대해서는 구체적 행동을 decoder가 취하지 않는 것처럼 보였고, 정상적인 압축 블록들만을 가지고 decoding이 수행되었다. 본 연구 진행 시점 현재까지 NPNCB가 삽입된 압축 데이터에 대해 시중의 압축 프로그램(반디집 7.27-빌드#54034,2022/07/07,x64 및 알집 12.10.0.2)과 백신 프로그램(V3, 알약)에서는 이와 같은 NPNCB injection을 감지하지 못했으며, 원본 데이터 없는 비 압축 블록이기 때문에 일반적인 Deflate decoder의 경우 압축 해제 시, NPNCB로부터 어떠한 압축 해제 데이터를 생성하지 않고 넘어가므로, 앞서 언급한 대로 압축 해제된 데이터만을 가지고 CRC 체크를 하는 기존 압축 프로그램들의 압축 데이터 변조 여부 검증 방법을 회피할 수 있었다. 또한 NPNCB의 DBA 영역에 악의적인 데이터가 추가된 경우 또한 감지되지 않았으며, DBA영역의 데이터에 대해서 기존 압축 프로그램들은 검증을 하지 못하고 정상적으로 압축 해제 되었다.

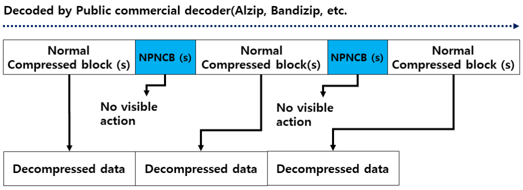

Fig. 12. Decoding the malignant NPNCB block injected compressed data by normal compression program

1.2절에서 언급한 바와 같이 DBA영역은 그 크기와 시작점이 앞선 압축 데이터 블록의 끝나는 위치에 따라 byte align을 위해 필요한 만큼으로 결정되므로 동일한 악성 코드조차 바이트 단위로 살펴보면 원본 데이터의 압축 결과에 따라 NPNCB의 시작 위치가 달라지므로, 동일한 악성 코드도 바이너리 형태가 다양한 형태로 변형되어 압축 데이터에 포함될 수 있다. 마지막으로 NPNCB내에서도 DBA에 기록되는 악의적 코드 자체도 변조할 수 있기 때문에 따라서 기존 백신 프로그램으로 해당 악성 코드의 탐지가 매우 어렵다.

2.4 No-payload non-compressed block 삽입 공격을 통한 임의 코드 실행 개요

2.1절에서 언급한 바와 같이 NPNCB를 삽입할 때, DBA 영역에 특정 악성 코드를 함께 탑재하여 삽입이 가능하다. Fig. 13.에서와 같이, 일반적인 decoder로는 무시되는 NPNCB 블록에 대해서 악

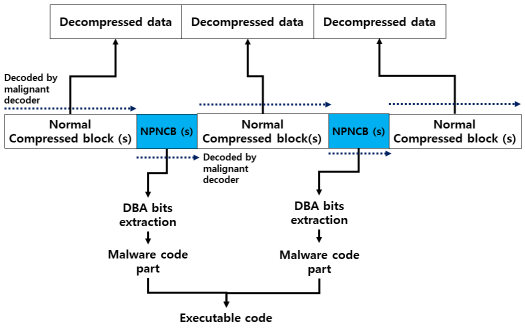

Fig. 13. Decoding the malignant NPNCB block injected compressed data by malignant decoder

의적인 decoder는 각 NPNCB를 압축 해제 하면서, 각 DBA 영역의 데이터를 추출하여 미리 설정된 시나리오에 따라 재조합하여 악성 행위가 가능한 형태의 코드를 생성한 뒤 이를 직접 실행하거나 또는 다른 악성 행위와 결합되어 최종적인 공격을 수행할 수 있게 된다.

2.5 정상적 압축 데이터에 대한 개별적 NPNCB 삽입 공격

이와 같은 취약점의 또 하나의 잠재적 위협은 악의적 encoder가 압축을 할 때 압축 블록을 생성함과 동시에 NPNCB를 삽입 하는 경우뿐만 아니라, Fig. 14.와 같이, 일반적인 압축 프로그램으로 압축된 정상적인 압축 데이터에 대하여도 “시차를 두고” 우연한 기회를 통해 오염시킬 압축 파일에 접근하여 악의적 코드(malware)가 정상적인 압축 블록들 사이의 경계 지점을 식별하여 어떤 위치에든 원하는 곳에 DBA 영역에 악성 코드가 탑재된 NPNCB를 삽입할 수 있고, 이렇게 오염된 압축 데이터는 일반적인 압축 프로그램으로 정상적으로 압축해제 됨에 있다. 따라서 본 취약점은 겉모습은 다른 파일 형식이지만,

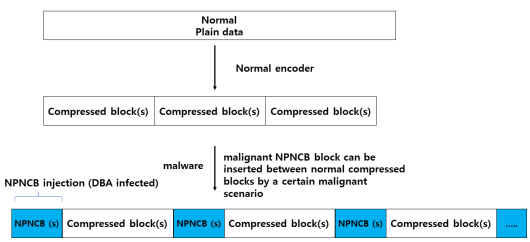

Fig. 14. Malignant NPNCB block can be inserted between normal compressed blocks by designed attack scenario

실질적으로 Zip 형식을 채택하고 있는 안드로이드 실행 파일(.apk) 및 엑셀 이진 압축 파일(.xlsb)등 다양한 파일에 적용되며, 추가 연구가 필요하겠지만, 내부적으로 Deflate 알고리즘이 사용된 .jpeg, .png, .pdf 등에도 광범위하게 적용될 것으로 예상된다.

III. 공격 사례 구현

3.1 악의적 실행 코드 삽입 개요

본 연구에서는 NPNCB를 정상적인 테스트 파일을 압축하면서 동시에 생성된 압축 블록사이에 임의로 삽입하여 생성한 오염된 Zip 파일을 악의적 압축 해제프로그램으로 압축 해제하면서 직접 NPNCB들의 각각의 DBA로부터 악성 코드를 조합하여 실행하는 모의 공격 프로그램을 작성하였다.

악의적 코드는 Fig. 15.과 같은 화면을 생성하는 7680 바이트의 Windows용 응용 프로그램으로서 버튼(button1)을 클릭하면 간단한 팝업이 나오는 테스트 코드로 하였다. 구현의 용이와 공격 개요를 쉽게 표현하기 위해, NPNCB는 Fig. 16.과 같이 DBA에 악성 코드를 담은 NPNCB들을 정상적인 압축 블록의 맨 앞에 연속적으로 삽입하여 변조된 압축 데이터를 만든 뒤, 일반 압축 SW 로 압축 해제한 결과와 본 연구에서 취약점 테스트를 위해 개발한 간단한 악의적 행동을 하는 압축 SW로 압축 해제 하여 보았다.

Fig. 16.을 좀 더 상세히 Fig. 17.에 표현하면, 위에서 언급한 7680 바이트의 악성 윈도우 프로그램 코드의 각 1 바이트를 5비트와 3비트로 분리하여 2개의 NPNCB의 DBA에 각각 나누어 담았다. 2개

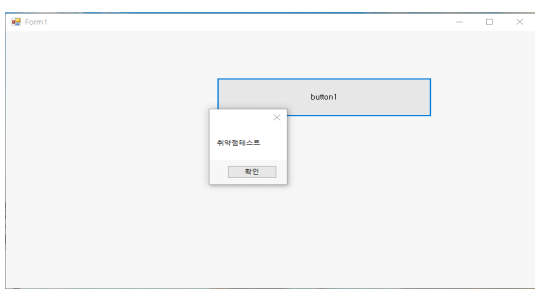

Fig. 15. The running result of test malignant code(TEST_SECURE.exe) which will be fragmented into NPNCBs' each DBA

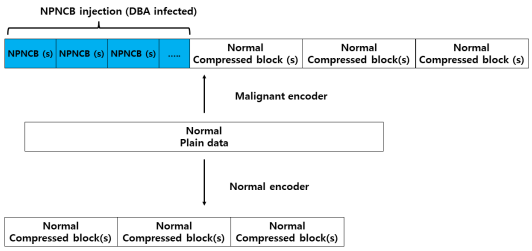

Fig. 16. NPNCB injection attack test case by malignant decoder compared with normal encoder

의 NPNCB의 DBA 영역에 악성코드 1 바이트를 5비트, 3비트로 나누어 담았기 때문에, 전체 NPNCB의 필요 개수는 7680(악성코드 바이트 수) × 2 개 = 15360 개이다. Fig. 4.와 같이 1개의 NPNCB가 바이트 경계에서 시작할 경우에는 5바이트를 차지하므로 15360개의 NPNCB × 5바이트(NPNCB 1개당 차지하는 바이트 수) = 76800 바이트의 악성 데이터가 압축 데이터의 앞부분에 추가되게 된다. 주의할 점은 NPNCB가 바이트 경계에서 시작하지 않을 때에는 반드시 크기가 5바이트인 것은 아니라는 점이다. 본 연구에서는 NPNCB를 바이트 경계에서 시작하도록 하는 단순한 공격 시나리오를 채택했기 때문에 1개의 NPNCB가 5바이트를 차지하게 된 것이다.

또한 본 연구의 테스트 악성코드는 .net framework 으로 개발된 상대적으로 방대한 코드이므로, C 또는 어셈블리어로 경량화 하여 제작되거나 간단한 스크립트 코드라면 훨씬 작은 용량을 차지할 것으로 예상된다.

실제로 Fig. 15.에서 소개한 악성 코드를 주입한 Zip 파일 생성 결과는 Fig. 18.의 TEST_ZIP.zip 이며 이를 이스트 소프트사의 알집 프로그램으로 압축 해제 시 정상 압축 해제되어 담겨있던 정상 원본 데이터인 '압축해제화일.xlsb'를 정상적으로 생성함을 확인할 수 있다. 그림에는 표현되지 않았지만 반디집(BandiZip) 과 Winzip의 경우도 같은 결과였다.

이제 주입된 악성 코드의 실행을 테스트하기 위해 동일한 Zip 파일을 본 연구에서 테스트를 위해 개발한 악의적 decoder로 압축 해제 시 Fig. 19.같이 압축 해제와 동시에 원본 데이터인 '압축해제화일.xlsb' 는 정상적으로 압축해제 되었고 이어서 DBA 로부터 추출한 악성 코드도 다시 재조합 되어 메모리상에서 실행되면서 아래와 같은 윈도우 창이 자동적으로 실행되어 나타남을 확인할 수 있었다.

3.2 공격의 한계점 및 잠재적 위험성

비록 Deflate 알고리즘이 매우 광범위하게 사용되고 있고, 오염된 압축 데이터를 생성하는 과정은 매우 용이하나, 공격을 유발하기 위하여 악의적인 decoder가 압축을 해제하면서 NPNCB의 DBA영역에서 주어진 시나리오에 따라 악성 코드를 재조합하여야만 하는 장벽이 있다. 그러함에도 SFX(Windows self-extracting archive) 형식과

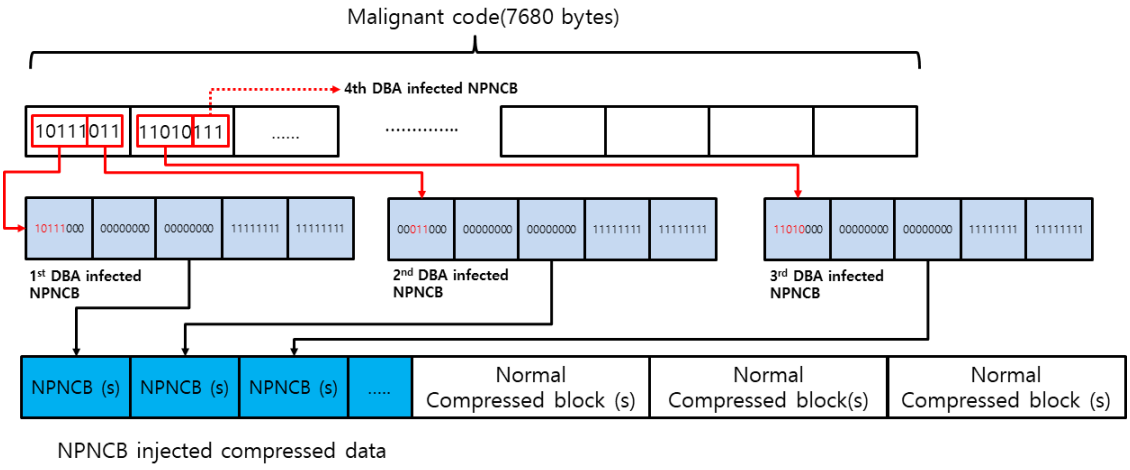

Fig. 17. The summary of malignant code dividing into each NPNCB DBA in the test attack case

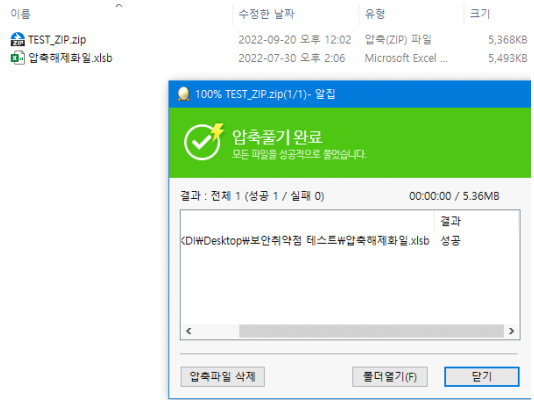

Fig. 18. Malignant zip file decompression success by normal decoder

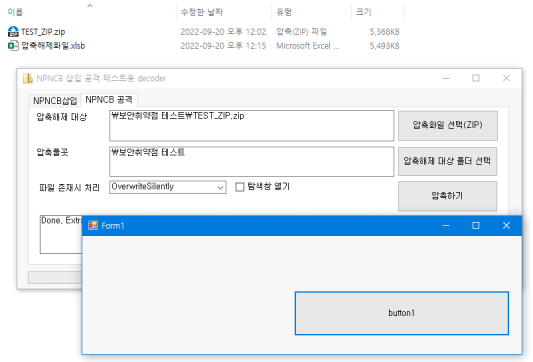

Fig. 19. Malignant zip file decompression success by malignant decoder

같이 실행형 decoder 가 악의적일 경우 NPNCB 삽입 취약점에 대한 패치나 보안 조치가 없는 경우 SFX 파일 실행으로 삽입되어 있던 NPNCB 로부터 악의적 코드가 재조합되어 실행될 수 있다.

또한 간단한 NPNCB삽입 과정을 통해 오염된 데이터를 Deflate 압축 블록내에 숨겨서 정상 압축 데이터로 위장하여 보안 장벽을 뚫고 들어갈 수 있으며, 그 자체가 일종의 스테가노그래피(Steganography) 기술로서 활용되어 새로운 데이터 숨김 장소를 제공할 수 있으므로, 직접 공격 수단이 아니더라도 중요 보안 데이터를 손쉽게 숨겨서 내/외부로 이동시킬 수 있는 잠재적 위험 또한 존재한다.

참고로, Zip 형식과 관련한 추가적인 취약점으로는 본 연구에서 주로 다룬 Deflate 알고리즘 자체의 취약점 보다는 Zip 파일 구조 자체의 취약점으로 인한 데이터 숨김 가능성[7] 및 일부 디코더들의 폴더 검증 취약점을 이용한 Zip-slip 취약점이 알려져 있다[8].

IV. 결 론

본 연구에서는 매우 대중적인 압축 알고리즘인 Deflate 에서 허용하는 3가지 종류의 압축 블록 중 non-compressed block의 특수한 유형인 원본 데이터 없는 비 압축 블록(No-Payload Non-Compressed Block;NPNCB)의 DBA 영역을 악성 코드로 오염시킨 뒤, 정상적인 압축 데이터 블록 사이에 실제로 삽입하였고, 현재 까지 나온 상용화된 압축 프로그램에서 아무런 경고 없이 원본 데이터가 정상 해제 되었으며, 악의적 디코더로 동일한 압축 파일로부터 숨겨진 악성 코드를 임의 실행 가능함을 구체적으로 보였다. DBA는 Deflate의 특성상 동일 악성 코드라도 원본 데이터가 달라질 경우 직전 블록까지의 압축 결과에 따라 가변적으로 크기가 변화하므로 오염된 압축 데이터의 바이너리(binary) 값 자체가 달라지는 특성이 있을 뿐만 아니라, DBA 데이터 자체를 여러 단계로 추가 변조할 경우 그 탐지가 더욱 쉽지 않으며 또한 본 연구가 진행된 현재까지 어떠한 압축 프로그램에서도 이와 같은 취약점을 경고하지 못하고 있다. 따라서 본 논문에서는 이와 같은 취약점을 구체적으로 보이고 조속한 보안 대책의 필요성을 알리고자 한다.

References

[1] P. Deutsch,Aladdin Enterprises, "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951,May. 1996.

[2] J. Woods, Proteon, Inc., "PPP Deflate Protocol",RFC 1979,Aug. 1996.

[3] S.Hollenbeck,VeriSign, Inc., "Transport Layer Security Protocol Compression Methods", RFC 3749,May 2004.

[4] T. Ylonen, SSH Communications Security Corp,C. Lonvick, Ed., Cisco Systems, Inc.,"The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, Jan. 2006.

〔5〕 Zlib 1.2.11 manual,“Zlib manual”, https://www.zlib.net/manual.html,27th Sep. 2022.
〔6〕 .Zip File Format Specification,“Zip appnote”, https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT,27th Sep. 2022.
〔7〕 Steganography with zip archives,“Zip Steganography”, https://github.com/gromnitsky/zipography,27th Sep. 2022.
〔8〕 Zip Slip Vulnerability,“Zip slip”, https://security.snyk.io/research/zip-slip-vulnerability,27th Sep. 2022.

〈저자소개〉

김 정 훈 (Jung-hoon Kim) 정회원
2002년 2월: 서울대학교 약학과 졸업
2010년 2월: 서울대학교 보건대학원 보건학 석사 졸업
2013년 2월: 서울대학교 보건대학원 보건학 박사 수료
2019년 2월: 경희대학교 컴퓨터공학과 SW융합학 석사 졸업
2016년~현재: 바이너리랩 주식회사 대표이사
〈관심분야〉 의료정보, 정보이론, 정보보호