날이 갈수록 보안은 중요해지고 있습니다.

특히 웹사이트는 많은 사람과 통신하는 경우가 많아서 보안이 더욱 중요합니다.

웹사이트를 운영, 개발하시는 분들은 단순히 외부 보안 솔루션에 의존해서는 안되고, 스스로 보안에 대한 이해를 쌓아가셔야 다양한 보안 위협으로부터 유연한 대처를 할 수 있습니다.

보안을 평가하는 기준은 매우 엄격합니다.

보안이 가장 약한 곳이 전체의 보안 수준을 결정합니다.

화려한 프론트엔드와 뛰어난 백엔드 로직으로 이루어진 웹사이트이더라도 보안면에서 문제가 있으면, 많은 사용자에게 신뢰를 잃을뿐더러 법적으로도 이런 웹사이트는 운영해서는 안됩니다.

웹 개발자라면 꼭 알아야 할 보안 기초

당부의 말

이 가이드북을 통해 여러분의 웹 애플리케이션 코드가 더욱 견고해지고, 코드를 더 잘 분석하고, 코드를 작성할 때 더 신중하게 된다면, 이 가이드북의 역할은 다한 것입니다. 비평은 언제나 환영입니다. 비평을 기다리고 있겠습니다.(Request for comment)

외부 공유 및 활용은 언제나 환영입니다.

보안은 정말 범위가 넓습니다. 시스템 보안, 네트워크 보안, 사물인터넷 보안, 인공지능 보안, 애플리케이션 보안 등등 보안의 세계는 정말 넓고, 점점 더 확장되어 가고 있습니다. 여기에서는 평범한 웹 개발자들이 더 나은 웹을 만들어 나가는 데 필요한 보안의 기초적인 부분에 대해 알아봅니다. 보안에 대한 기초적인 상식을 기르고, 앞으로 여러분의 서비스에 보안 조치 작업을 직접 행동으로 옮길 수 있게 하는 게 목표입니다.(어려운 말들로 가득한 보안 매뉴얼에 따라서 하는 수동적인 작업이 아닌, 능동적으로 보안을 구성해나가는 것)
웹에서 보안 공격이 왜, 어떻게, 어디서 일어나는지 알아보고, 이에 대처하며 웹을 만들어가는 방법을 알아봅니다. 구체적으로 ***backend, frontend,*** 서버 운영체제에서 일어나는 공격과 이에 대처하는 방법, 웹에서 암호의 필요성과 암호의 구체적 이해, 더 안전한 코드를 작성하는 방법*(*시큐어 코딩*)*들로 전개됩니다.
깊게 들어가는 내용이 분명 있습니다. 그러나 이는 귀찮은 암기보다는 원리를 이해하며 알아가기 위해 넣은 것이니 일반 웹 개발자분들이라면 가볍게만 훑어보고 넘기시면 됩니다.[1]
웹 개발과 크게 관련 없는 내용도 있습니다. 세밀하게 학습하기 위해 넣었다는 것 보다는 여기서 얻을 수 있는 교훈을 위해 넣은 것이니, 이런 내용들은 간단하게 훑어보시면 됩니다.
아마 주로 네트워크 보안과 애플리케이션 보안, 암호 시스템의 비중이 높겠지만, 이것들만을 다루는 것은 아닙니다. 어떤 Backend, Frontend 기술을 사용하던, 공통적으로 가져야하는 보안의 시스템에 대해 다룹니다.[2]

---

[1] 정보보안기사를 준비하고 계시거나, 보안 전문가의 기초를 다지고 계신 분이라면 세밀하게 알아가시는 걸 추천합니다.
[2] Python과 Flask 웹 프레임워크를 이용해 실습을 진행합니다만 전체적인 내용은 특정 언어와 프레임워크에 종속되지 않도록 구성하였습니다.

들어가기 전에

보안에 대한 기본적인 오해

- 규모(회사 규모나 개발 규모)가 작은 웹 서비스에도 공격이 행해집니다. 하물며 바이러스 같은 건 일반 컴퓨터 사용자에게까지 크게 전파됩니다. 당연히 불특정 다수에게 공개되는 웹 서비스는 당연히 공격을 받습니다.
- FrontEnd에도 보안 공격이 일어납니다. 개인적으로 FrontEnd, BackEnd 중 보안에 더 신경써야 하는 분야를 고르라고 하면, 아마 못 고를 겁니다. 어느 개발 분야이든 보안은 다 중요합니다.
- 많은 사람이 사용한다는 게 안전하다는 보장은 아닙니다.
- 웹 서비스에 공격을 하는 사람은 정말 상상도 못한 방법을 동원할 때도 있습니다.[3]
- 내부자에 의한 보안 사고가 꽤 발생합니다. 고의적으로 사고를 일으켰던, 단지 개발과정에서 스파게티 코드로 인한 의도하지 않은 정보 유출 등. 특히 의도하지 않은 정보 유출은 당장은 눈에 보이는 피해가 없더라도 나중에 공격을 받았을 때 피해를 크게 증폭시키게 됩니다.
- 프로그래밍을 못하는 해커도 존재합니다. 자동화 툴이 많이 발달되어 있는 것 같습니다.

[3] 기술적인 외에도 포함됩니다.(운영자 고문拷問 등)

보안이란 무엇인가?

일반적으로 정보 보안은 3가지 중요한 요소를 가지고 있다고 말합니다.

1. 무결성

권한을 가진 사람만 정해진 방법으로 정보를 변경할 수 있어야 한다.
나의 개인정보를 변경할 수 있는 사람은 일반적으로 나 자신뿐입니다. 모르는 사람이 갑자기 나의 이름(id)과 비밀번호를 바꿀 경우 무결성이 침해된 겁니다. 민감한 정보뿐만 아니라, 모두에게 공개되는 나의 닉네임을 누군가 함부로 바꾸는 경우 등에도 무결성이 침해된 겁니다.

2. 기밀성

권한을 가진 사람만 정보에 접근할 수 있어야 한다.
나의 금고를 열 수 있는 건 나 자신뿐입니다. 원하지 않는 사람이 내 금고 안에 있는 내용(개인정보, 비밀번호 등)을 보는 것은 기밀성이 침해된 겁니다. 위에서 나온 경우처럼 모르는 사람이 나의 개인정보를 함부로 변경할 때, 개인정보를 보면서 변경하였다면 기밀성까지 같이 침해된 것으로 볼 수 있습니다.

3. 가용성

(권한이 있을 때) 원할 때 정보에 대한 접근이 가능해야 한다.
필요한 상황에서 나의 금고는 언제나 열릴 수 있어야 합니다.(나에 의해서) 내가 나의 금고를 열려고 하는데, 안 열리는 경우에는 가용성이 침해된 겁니다. 일상생활에서 24시간 언제든 접근 가능한 곳(편의점) 등은 가용성이 매우 크다고 볼 수 있습니다. 편의점에 사람이 너무 많아 못 들어가는 경우에는 일상생활에서의 가용성이 침해된 겁니다.

웹 보안의 구성

웹 보안의 구성이라고는 하지만, 웹 개발분야가 아닌 개발 분야에서의 보안 구성과 겹치는 부분이 많습니다. 가볍게 웹 보안 구성 요소에 대해 알아봅시다. 보안 구성을 분류하는 기준은 다양합니다. 아마 인터넷에서 웹 보안 구성을 찾아보시면 이 가이드북과 살짝 다를 수 있습니다. 웹 보안 구성은 네트워크 패킷의 이동 경로에 맞추어 설명하기도 합니다. 그러나 본질적으로는 같은 이야기를 하고 있는 것이니 참고해주세요.

1. 방화벽

이상한 네트워크 패킷을 원천적으로 차단해주는 역할을 합니다. 일반적인 웹 서비스라면 tcp 80번, tcp 443번을 제외하면 공개해야할 포트가 많이 없을 겁니다. 이러한 방화벽은 애플리케이션(웹 서비스 자체)에 대한 공격은 못 막습니다.

2. 시큐어 코딩

프로그래밍을 할 때 보안에 신경써서 제작하는 게 중요합니다. 취약점이 될 수 있는 논리적 오류 등을 개발 단계에서 개발자가 신경쓰는 개발기법입니다.

3. 데이터 보안

DB에 저장되어 있는 데이터에 대한 보안에도 신경써야 합니다. 특히 DB 서버와 애플리케이션 서버가 분리되어 있는 경우에는 더 관심을 가지고 관리해야 합니다.

4. 로그 관리

웹 서버의 로그가 잘 기록되도록 해야 합니다. 로그를 분석하면서 공격이 이루어진 정황을 찾아낼 수 있습니다.

5. 접근 관리

웹 서버에 접근하는 것에 대한 관리도 필요합니다. root나 administrator 계정이 유출되지 않도록 주의합니다. 이를 위해서는 안전한 운영체제를 사용하고, 악성코드 같은 것들을 주의해야 합니다.

1, 4, 5번 항목은 2번과 3번 항목에 비해 신경쓸 것이 적다고 생각합니다. 그래서 이 가이드북에서는 2번과 3번 위주로 내용이 진행됩니다. 그렇지만 방화벽, 로그, 접근 관리를 무시해도 되는 건 아닙니다.

암호의 기본 개념

암호의 활용 가능성은 무궁무진합니다. 세세한 내용을 다루기 전에 간단한 용어들을 먼저 정리해봅시다.

- 평문

암호화되지 않은 메시지(데이터) 원본. 영어로는 plaintext.

- 암호문

평문을 암호화했을 때 메시지(데이터). 영어로는 ciphertext.

- 암호화

암호 알고리즘(함수)를 사용해서 평문을 암호문으로 바꾸는 것. 영어로는 encryption.

- 복호화

암호 알고리즘(함수)를 사용해서 암호문을 평문으로 되돌리는 것. 영어로는 decryption.

- 암호해독

암호 시스템을 무력화 시킨다는 것. 암호문으로부터 평문에 대한 정보를 알아내는 것.

만약에 key가 없다면, 암호 알고리즘을 알고만 있어도 암호화와 복호화를 할 수 있을 겁니다.
어떤 두 사람만 특정 메시지를 보고자 할 때는 두 사람만 알고있는 key를 이용해 암호 알고리즘을 사용하면 됩니다.
메시지를 암호화할 때는 다음과 같은 표현식으로 나타낼 수 있습니다.
Encryption(Plaintext, Key)
메시지를 복호화할 때는 다음과 같은 표현식으로 나타낼 수 있습니다.
Decryption(Ciphertext, Key)

이러한 구조는 보통 대칭키 암호 구조입니다. 간단한 대칭키 암호는 인류 역사상 아주 오래되었습니다.
비대칭키 암호는 1970년대에 처음 나온 방식입니다. 암호화를 할 때 사용하는 key와 복호화를 할 때 사용하는 key가 다르고, 이를 이용해 다양한 활용을 할 수 있습니다.

보안 공격

이제부터 웹 서비스를 위협하는 보안 공격들에 대해 알아봅니다.
정말 다양한 유형의 공격이 지금 인터넷 어디에선가 일어나고, 새로 만들어지고 있습니다. 많은 공격들이 현재 다양한 보안패치, 운영체제 내에서의 보안 작용으로 큰 효과를 내지 못하고 있습니다. 그러나 이 공격의 원리에서 얻을 수 있는 교훈은, 새로 나오고 있는 보안 공격에 대처하는 핵심을 제공합니다. 다양한 보안 공격을 알아보고 이에 대응하는 원리를 알아가봅시다.

- 서비스 거부 공격(DOS)

말 그대로 다른 사람이 서비스를 받지 못하게 방해[4]하는 공격입니다. 즉, 가용성을 공격하는 것이죠. 공격자가 시스템의 자원[5]을 모두 소진시켜서(사용 또는 파괴) 정상적인 사용자가 시스템을 정상적으로 이용할 수 없게 하는 겁니다. 흔히 말하는 과부하가 걸리게 하면 되는 공격이기에 많은 종류가 있고, 자주 발생합니다. 보통 테러의 성격을 가지고 있습니다.

취약점 공격 **DOS:**
일반적으로 네트워크로 한 번에 데이터를 보내지 않습니다. 한 번에 전송될 수 있는 데이터는 기본적으로 1500byte 정도입니다. 그 이상의 데이터를 보낼 때에는 사실 데이터가 여러 번 전송되는 겁니다. 근데 이때 데이터의 순서가 뒤바뀌어서 목적지에 도착할 수 있으니,
데이터(패킷)을 보낼 때, 앞에 순서를 붙입니다.[6] 근데 이때 공격자는 데이터를 보내는 데, 이걸 조작하여 순서가 불완전하도록 하는 겁니다.

정상적인 경우

도착 순서	패킷에 적힌 데이터 순서
1	2
2	1
3	3
4	4

[4] 영업 방해라고 생각하시면 됩니다.
[5] 어려운 용어가 아닙니다. 서버의 cpu, ram 점유율 및 서버 컴퓨터와 네트워크 간에 동시에 연결할 수 있는 양 등을 의미하는 겁니다.
[6] 참고로 이 기능은 http의 기반인 tcp 프로토콜의 주요 특징입니다. udp 프로토콜은 복잡한 것 없이 오는 순서대로 합칩니다.

이런 경우 아무 문제 없습니다. 순서가 잘 지정되어 있으니 이에 맞추어 데이터를 받은 컴퓨터가 다시 합쳐주면 되니까요!

공격자

도착 순서	패킷에 적힌 데이터 순서
1	1
2	1
3	3
4	5

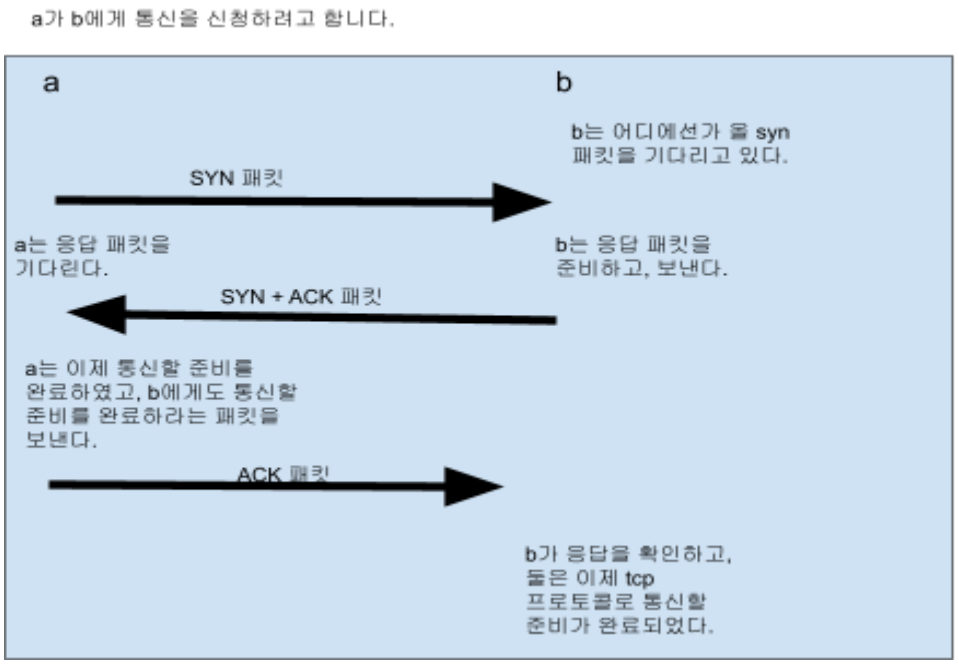
공격자는 이렇게 데이터 순서가 중복되거나, 순서 사이에 빈 것이 존재하도록 패킷을 만들어서 공격 대상에게 전송합니다. 공격 대상은 해결할 수 없는 문제를 해결하고자 무한히 시도하다가 과부하가 걸립니다.

이외에도 취약점을 이용한 공격은 더 있습니다.
패킷을 보낼 때, 패킷을 보낸 주소(응답을 보내야 하는 주소)를 패킷의 도착 주소와 같게 조작하여 보내는 겁니다. 이 패킷을 차단하지 못하면, 패킷의 응답을 자기자신에게 보내게 됩니다.
시스템에 따라서는 무한루프에 빠져 오류를 일으키는 경우, 자기자신에 대해 세션을 생성하는 경우가 발생할 수 있습니다. 즉, 자기자신(공격자가 조작한 주소)와 자기자신(패킷을 받은 주소)끼리 정상적인 tcp 연결을 해버리는 겁니다. 이 과정에서 컴퓨터의 자원은 소모되고, 결과적으로 서비스 거부가 이루어져 버리게 되는 겁니다.

첫 번째 유형은 과부하가 예상되는 패킷은 따로 분리해두었다가 폐기하고, 두 번째 유형은 패킷을 처리하기 전에 먼저 패킷을 보낸 주소가 정상적인지 확인하면 됩니다.
이런 취약점 공격은 아주 간단하게 방어할 수 있습니다.
안정된 버전의 최신 운영체제(+보안 업데이트)를 쓰면 자동적으로 거의 완전한 방어가 가능합니다. 취약점에 그대로 당하는 오래된 운영체제라면, 하나하나 이런 공격의 방어를 구현해주어야 합니다.
보안 측면에서는 오래된 시스템(운영체제)는 좋은 선택이 아니라고 판단해서 취약점 공격 관련 내용을 넣게 되었습니다.

SYN Flooding:
tcp는 양방향을 지향합니다.

tcp로 통신을 시작할 때 일어나는 과정을 간단하게 살펴보겠습니다.[7]

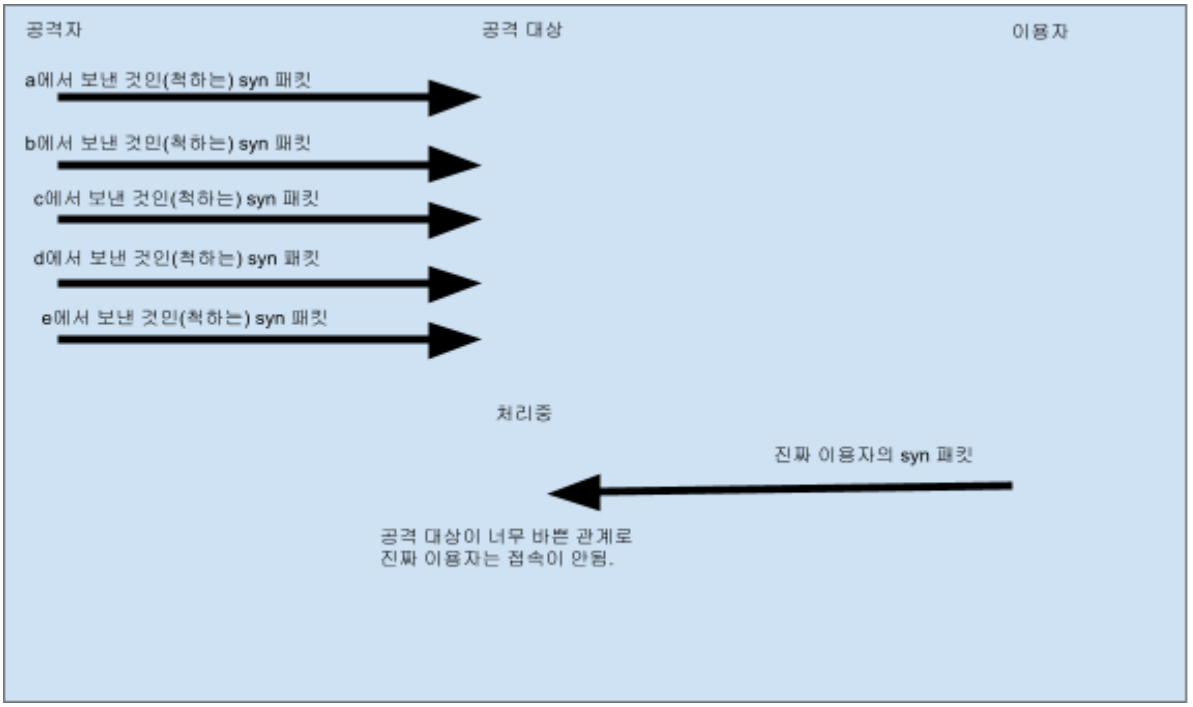


여기서 a가 syn 패킷을 보낼 경우, b가 이것을 받고, syn+ack(a에게 syn 패킷을 받았다는 ack 응답 패킷, a도 통신 준비가 완료되었는지를 물어보는 syn 패킷)을 보냅니다. 그리고 a에게서 올 예정인 ack 패킷을 기다립니다.

여기서 공격자는 ack 패킷을 보내지 않습니다. 엄밀히 말하면, 출발지 주소를 존재하지 않는 곳으로 위조하기 때문에 ack 패킷을 받을 수가 없습니다. 위의 상황에서라면 b는 하염없이 a에게 올 예정인 ack 패킷을 기다리게 됩니다. 너무 오래기다가 timeout이 되어 자동적으로 통신 연결이 끝납니다. 이때 timeout이 될 때까지 컴퓨터 자원이 소모 됩니다.

물론 1번으로는 아무 영향이 없고, 공격자는 서로 다른 곳에서 보낸 척하는 syn 패킷을 동시에 공격 대상에게 보냅니다. 공격 대상은 서로 다른 곳에서 온 것으로 착각하고, 이에 맞추어 하나하나 통신을 준비합니다.(연결 준비 과정을 패킷이 온 만큼 하는 겁니다)
그런 과정에서 극심한 시스템 낭비가 발생하고, 서비스 거부 상태가 되는 겁니다.

[7] 모르는 분들이 있으실 것 같아서 간단하게 넣었습니다.



이런 공격은 대응이 간단해 보일 수 있습니다. timeout 되는 시간을 짧게 줄여서, 최대한 빨리 공격 패킷이 무효화되길 기대할 수 있습니다.
단순해 보이는 대처 방법인 만큼 잘못된 대처 결과를 가져올 수 있습니다. 너무 시간을 짧게 줄여버리면, 데이터 전송이 조금 느린 컴퓨터를 가진 일반 사용자도 접근이 거부될 수 있습니다. 보안 관점에서는 괜찮을 지 몰라도, 사용자 경험이 안 좋아질 수 있습니다.

지금은 간단한 tcp 통신 시작 과정에 대한 이야기이지만, 웹과 직접적인(frontend, backend) 부분에서도 비슷한 상황은 충분히 생길 수 있습니다. 이부분에 대한 고민을 충분히 해보시면 사용자 경험과 보안, 두 개를 동시에 잡을 수 있게 될 능력을 기르실 수 있을 겁니다.

SYN Flooding의 경우, syn 쿠키라는 기술을 이용하여 효과적으로 방어할 수 있습니다.
공격 대상은 들어온 syn 패킷에서 연결이 정상적으로 되어 있을 때, 필요한 정보들을 쿠키 형태로 만들어 놓습니다. 그리고 다시 syn+ack 패킷을 보낼 때, 이 쿠키를 첨부하여 보냅니다. 이제 공격 대상은 더이상 syn 데이터를 메모리에 담아둘 필요가 없게 됩니다.
만약 정상적인 사용자라면 ack 패킷을 보내줄 때, 자신이 받은 이 쿠키를 다시 첨부하여 보내줄 것이고, 이를 이용해 정상적인 연결을 만들 수 있습니다.
만약 공격자가 공격 대상을 공격한다면, 어차피 바로 쿠키로 만들어 버려서, 메모리 소모가 적기 때문에 유의미한 공격을 할 수 없게 됩니다. 그러나 syn쿠키는 tcp 연결에서 기타 옵션을 설정한 것을 반영할 수 없게 되어서, 성능 저하 등의 문제가 생긴다고 알려져 있습니다.

보통의 경우(일반적인 운영체제)에서는 syn flooding이 의심되거나, 메모리 사용량이 크게 늘어날 경우 자동적으로 syn 쿠키가 적용됩니다.

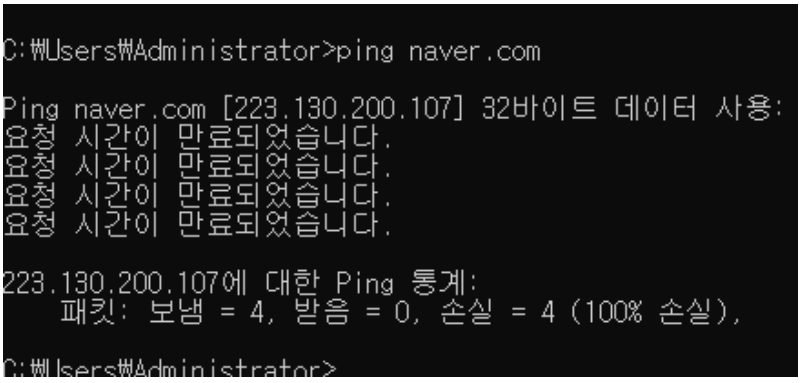
Ping of Death:

이 DOS는 ICMP를 이용한 공격입니다.

윈도우에서 흔히 사용하는 ping 명령이 icmp를 이용하는 것입니다.
icmp는 인터넷 제어 메시지 프로토콜로, 네트워크가 잘 작동하는지 체크해주는 기능을 수행합니다. 아주 간단한 역할을 수행하기 때문에 신경을 써야 할 필요가 있나 싶지만, 이를 악용하면 강력한 DOS를 할 수 있습니다.

공격자는 이 icmp 패킷의 크기를 매우 크게 조작하여, 동시에 여러 개를 공격 대상에게 보냅니다. 앞서 나왔던 것처럼 네트워크 패킷의 크기는 매우 작습니다. 공격자의 icmp 패킷은 이동을 하면서, 여러 개의 패킷으로 갈라지게 되고, 공격 대상은 받은 여러 개의 패킷을 다시 합칩니다. 이 과정을 많이 반복하다보면 시스템에 과부하가 걸리게 됩니다.

icmp 패킷을 이용한 공격은 시스템의 방화벽에서 icmp를 차단하면 끝나는 일입니다.



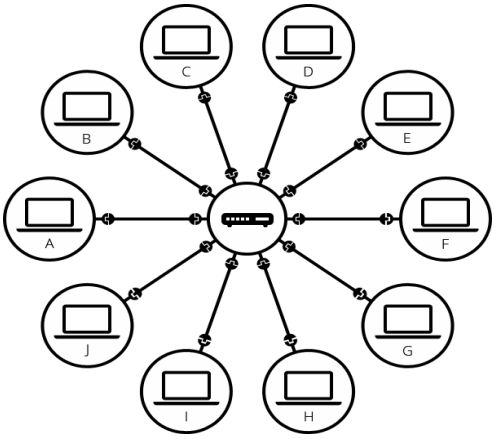
또한 일반적인 패턴의 죽음의 핑 공격은 최신 운영체제에서 자동적으로 방어할 수 있도록 되어 있습니다. 확실하게 웹 서버에서 확실하게 죽음의 핑 공격을 방어하고 싶다면, icmp를 차단시켜 놓으면 됩니다.

아주 사소한 기능, 동작이라도, 잘 이용하면 엄청난 피해를 줄 수 있습니다.

스머프 공격:

이 공격 또한 icmp를 이용한 공격입니다.

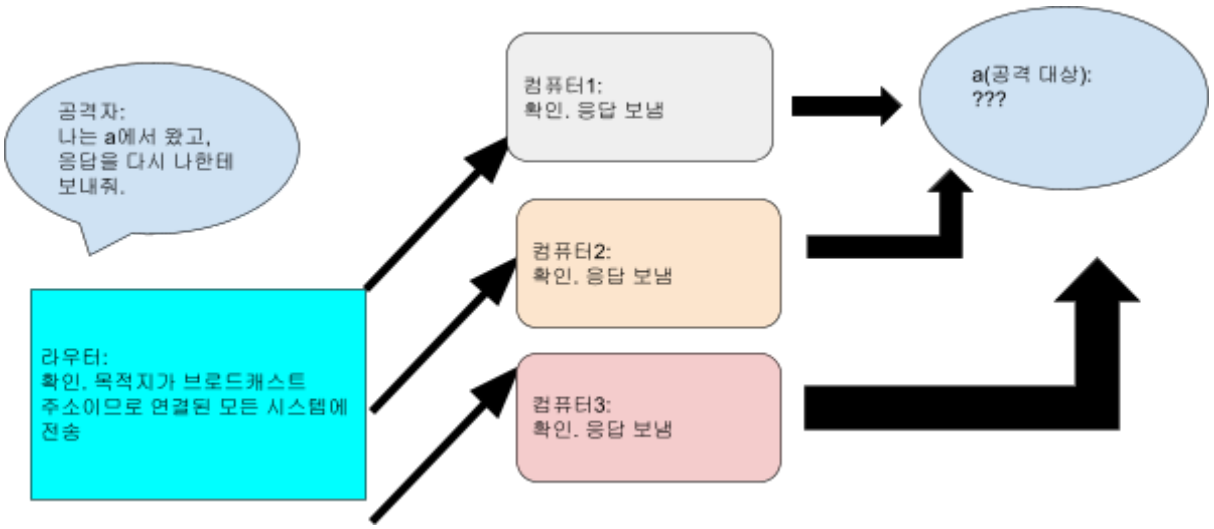
이 공격을 이해하기 위해서는 Broadcast 주소의 특징을 알아야 합니다. 브로드캐스트는 어떤 네트워크의 ip 주소 중 가장 큰 값을 말합니다. 네트워크 클래스에 따라 다르지만, 간단한 예시를 들자면 어떤 라우터에 연결된 장비들의 ip 주소가 1.1.1.1, 1.1.1.2, 1.1.1.3 이런 식으로 되어 있을 때, 브로드캐스트 주소는 1.1.1.255입니다.[8]



이 브로드캐스트의 역할은 연결된 모든 시스템에 패킷을 보낼 수 있다는 겁니다.
여기서 공격자는 출발지가 공격 대상인 icmp 패킷을 만듭니다. 당연히 도착지는 어떤 네트워크의 브로드캐스트 주소로 해서 보냅니다. 이 패킷은 브로드캐스트의 해당 네트워크 전체에 패킷이 퍼지게 됩니다. 각 시스템은 icmp에 응답하고자, 출발지로 적혀있는 공격 대상에게 응답을 보냅니다. 연결된 시스템이 많을 경우 위력이 커지고, 공격 대상은 원하지 않은 너무 많은 icmp 응답을 받고 마비되어 버립니다.

이 공격은 브로드캐스트의 역할을 제한하는 것으로 방어할 수 있습니다. 브로드캐스트 주소를 악용한 공격들을 막기 위해, 라우터에서 브로드캐스트 주소로 된 응답을 내보내는 것을 막는 겁니다. 그럼 이 공격을 수행할 때, 공격자가 보낸 icmp 패킷은 라우터와 연결된 시스템으로 보내지지만, 시스템에서 응답한 각 패킷들은 라우터에서 지나가지 못하도록 막는 겁니다.[9] 공격을 받는 쪽에서 할 수 있는 조치로는 죽음의 핑처럼 아예 icmp 패킷을 방화벽에서 막아버리는 것으로 방어할 수 있습니다.

[8] 주소 체계에서 들어갈 수 있는 최대 값은 255이니까요.
[9] 이 기능이 기본 설정으로 사용되는 경우가 많아지고 있습니다.



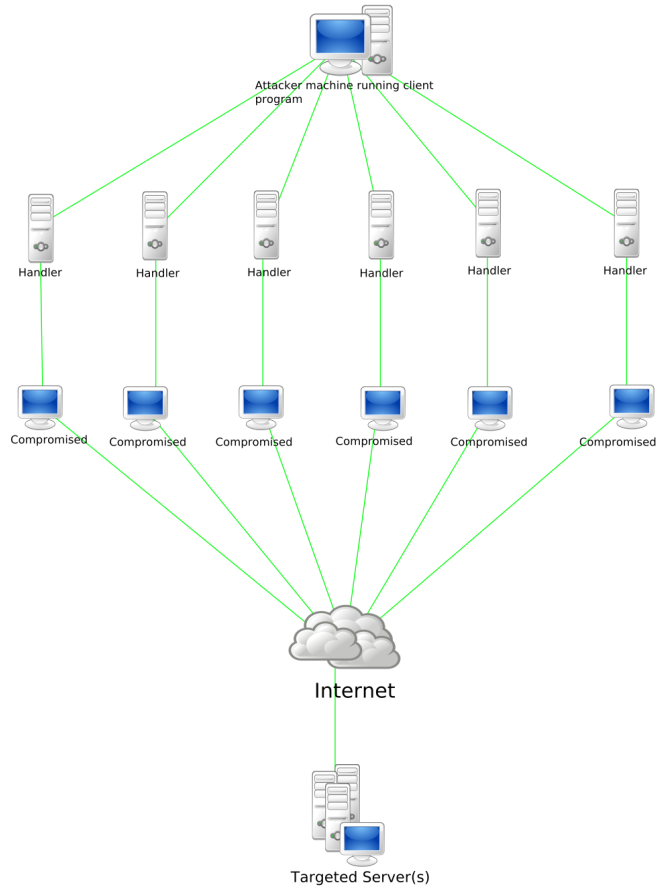
DDOS:

웹(http)과 큰 관련이 없고, 방어 대책도 쉽고 자동적으로 되어 있는 DOS들을 위주로 설명해보았습니다. 그러나 이 공격들에 사용되는 핵심 원리들은 알아둘 필요가 있습니다. 마지막으로 가장 중요하고, 위협적고, 자주 일어나는 DOS인 DDOS에 대해 알아보도록 하겠습니다.

분산 서비스 거부 공격으로 언론 등을 통해 가장 많이 알려진 DOS라고 생각합니다. 동시에 앞에 나온 DOS와는 비교도 안될 만큼 강력한 위력인 것 같습니다.

이 공격은 인터넷 트래픽의 폭주로 공격 대상이 정상적으로 네트워크가 작동하지 않게 하는 겁니다. DDOS는 1대 1로 싸우는 것이 아닌 1대 다수로 싸우게 됩니다. 일반적인 경우는 공격자가 다수의 보안이 약한 일반 사용자를 먼저 공격합니다. 그리고 그들의 컴퓨터에서 원격으로 진짜 목표에 트래픽을 보내도록 합니다.(좀비pc) 이로 인해 진짜 공격 대상은 너무 많은 요청으로 인해 정상적인 서비스를 하지 못하게 됩니다.

다같이 공격하기 위해 공격 날짜를 미리 정해야 합니다. 공격으로 사회적인 영향을 키우기 위해 특정 기념일에 공격하는 경우도 있었습니다.



좀비pc가 되지 않을려면

- 기본적인 보안 수칙을 지킵니다.(수상한 파일 실행하지 않기[10] 등)
- 웹 서버도 조심해야 합니다. 간단한 포트폴리오를 올려놓은 웹 서버도 약한 보안조치로 인해 좀비*pc*가 될 수 있습니다.

DDOS는 다른 DOS보다 넓은 정의를 가지고 있습니다. 그래서 다른 DOS, 보안 공격과 결합할 수 있습니다. 위에 소개한 다른 DOS를 좀비pc를 이용해 엄청나게 많은 양을 수행하면 DDOS가 되는 겁니다.

DDOS로 공격 대상을 공격할 때, 다양한 공격 유형이 있습니다.

1. 위에서 나온 DOS를 좀비PC들이 수행
2. 좀비PC들이 http 요청(GET, POST 등)을 연속적으로 보냄(https://inflearn.com/ 등의 요청을 계속적으로 보내기 등)

[10] 수상한 파일에는 ppt, docx, hwp 등도 포함됩니다. 오피스 파일에는 컴퓨터 시스템에 접근할 수 있는 매크로를 삽입할 수 있습니다.

    1) 특히 파라미터가 있는 경우, 이 파라미터를 적극적으로 이용합니다. 예를 들어 a, b 파라미터가 a^b 등 수학 연산을 한다면, a, b 파라미터 값에 아주 큰 수를 넣어서 지속적으로 보내는 것도 있습니다. 이외에도 정규표현식을 검사하는 로직에 검사에 시간이 걸리는 문자들을 넣어서, 서버의 자원을 낭비시키는 것도 있습니다.
3. 좀비PC를 실질적으로 만들지 않는 경우도 있습니다. 좀비PC는 아니지만, 웹 사이트 내부에 숨겨진 자바스크립트 코드가 실행되면서 자연스럽게 공격이 실행되는 경우가 있습니다. 뒤에 나올 Injection과 큰 연관이 있습니다.

갑자기 이슈가 되어 접속자가 많아진 웹사이트와 DDOS 공격을 받고 있는 웹사이트를 구별하는 것은 어렵습니다. 특히 숙련된 DDOS 공격은 알려진 해결책으로도 방어가 매우 어렵습니다. 뚜렷한 해결책이 없어서 ddos는 방어라는 표현보다는 완화라는 표현을 사용할 정도입니다.

갑자기 네트워크 처리량이 늘어나거나, cpu나 ram 사용량이 크게 늘어날 경우 DDOS로 의심해볼 수 있습니다. 다만 아까 나왔듯이 단순히 사용자가 몰리면서 생기는 것일 수도 있습니다. 만약 사용자가 몰릴 때 DDOS가 일어나면, 공격 완화도 어렵고, 많은 사용자들이 이용을 못하게 되므로 공격이 크게 작용할 수 있습니다. 공격을 받는 입장에서는 애매하고, 공격을 하는 입장에서는 매우 효과적으로 공격을 할 수 있기에 주의를 기울여야 합니다.

웹 개발 과정에서 비정상적으로 빠르게 같은 요청을 보내는 ip 등을 일시적으로 방화벽에서 차단하는 방법 등을 사용할 수 있습니다. 이외에도 로드 밸런서를 도입해도 효과적입니다. 로드 밸런서는 많은 사용자들이 몰릴 때를 대비해 많이 사용하지만, DDOS 완화에도 효과가 있습니다. 개발자가 직접할 수 있는 부분이고, 능숙하지 못한 공격에는 효과적으로 대응할 수 있습니다. 그러나 작정하고 공격할 경우, 단순히 개발자의 노력만으로는 힘듭니다.
원리에 대한 이해로 방어 능력을 키우는 것을 목적으로 하는 가이드북이지만, DDOS는 어쩔 수 없이 솔루션을 추천합니다. 보통 클라우드 컴퓨팅을 제공하는 곳에서 DDOS 완화 솔루션을 같이 제공하기도 합니다. 또한 DDOS 완화 서비스만 따로 판매하는 곳도 존재합니다.

- 코드 삽입 공격(Injection)

이 가이드북에서 다루는 보안 공격 중 가장 개발자에게 직접적인 보안 공격입니다.
https://owasp.org/www-project-top-ten/
주요 보안 공격 중 항상 최상위권을 유지하고 있는 공격입니다. 간단하게 공격을

요약하자면, 클라이언트에게 받은 데이터가 어떤 역할, 즉 실행가능한 코드가 담겨있고 이것이 서버 내부에서 실행되어 피해가 발생하게 됩니다.
이 공격의 방어는 웹 개발자의 역할이 큽니다. 정상적인 접속을 이루면서 몰래 공격 구문을 삽입하기 때문에 방화벽 등으로 걸러내는 것은 불가능에 가깝습니다.

SQL Injection:
코드 삽입 공격 중 가장 유명한 공격입니다. sql은 데이터베이스를 처리하기 위한 언어입니다.
backend에서 db에 있는 데이터를 처리하기 위해 sql로 질의를 합니다. 이때 질의를 하는 내용은 클라이언트(사용자)가 보낸 내용을 기준으로 처리될 때가 많습니다. 쉽게 말해, 로그인을 할 때 사용자가 입력한 정보를 바탕으로 계정을 찾기, 블로그 같은 서비스에서 사용자가 선택한 글을 삭제(또는 수정, 좋아요 등)하기 같은 겁니다.

어떤 서버에서 로그인 작업을 할 때 다음과 같은 과정이 이루어 집니다.

1. 사용자에게 입력 html을 보여준다.
2. 사용자는 로그인 정보를 적고, 서버로 제출한다.
3. 서버의 backend 코드는 sql로 데이터베이스에 질의를 한다.
4. sql의 내용은 데이터베이스에서 계정을 찾아라. ID는 '사용자가 입력한 ID' 그리고 PW는 '사용자가 입력한 PW' 이런 구조로 되어 있다.(미리 말씀드리자면 '따옴표 부분을 유의하여 보시길 바랍니다.)
5. 계정을 찾으면 그 계정과 관련된 다른 데이터도 사용자에게 보여준다.

만약 사용자가 사용자가 입력 id에 a를 넣고, 사용자가 입력한 pw에 1234를 넣으면, db에 있는 id가 a이고 비밀번호가 1234인 계정을 찾고, 찾으면 그 계정의 정보를 사용자에게 보여줄(frontend로) 겁니다. 데이터베이스 처리 구문은 다음과 같을 겁니다.

데이터베이스에서 계정을 찾아라. ID는 'a' 그리고 PW는 '1234'

이제 공격자는 약간의 허점을 이용할 겁니다.
ID 값에 따옴표를 포함할 것입니다.

데이터베이스에서 계정을 찾아라. ID는 '' ' 그리고 PW는 ''

ID 값에 따옴표를 포함하면서 ID를 검색하는 부분이 끝나버리게 되었습니다.
원래 개발자의 의도를 무시하는데 성공한 겁니다. 공격자는 admin이란 계정으로 로그인을 하고 싶어할 경우, 다음과 같이 ID와 PW 값을 넣으면 됩니다.

데이터베이스에서 계정을 찾아라. ID는 'admin' 뒤에 전부 주석' 그리고 PW는 '아무값'

id 값은 admin' 뒤에 전부 주석입니다. pw 값은 아무 것이나 넣어도 상관 없습니다. 이렇게 되면, 비밀번호를 검증하는 부분이 주석으로 인해 전부 무력화 됩니다. 이로 인해 결과적으로 id만 알면 계정을 찾아낼 수 있는 sql 구문으로 바뀌게 된 것이나 마찬가지입니다.
비밀번호 없이 관리자 계정으로 로그인이 되는 끔찍한 상황인 겁니다.

끔찍한 상황은 여기서 끝이 아닙니다. sql은 마지막에 ;세미콜론을 넣어 명령 구문이 끝났다고 표시해줄 수 있습니다.
다음과 같이 만들 수도 있습니다.

데이터베이스에서 계정을 찾아라. ID는 '아무값';
데이터베이스를 삭제하라 뒤에 전부 주석' 그리고 PW는 '아무 값'

조금 복잡해 보이지만, id 값은 아무값'; 데이터베이스를 삭제하라 뒤에 전부 주석이고 pw는 아무값입니다. 보면 알 수 있듯 계정을 찾으라는 구문은 사실상 아무 의미가 없고, 공격자의 입력으로 생긴 데이터베이스 삭제 구문이 핵심입니다. 이렇게 다양한 명령을 삽입할 수 있습니다.

일단 특수문자 필터링을 통해 공격을 막아낼 수 있습니다. 따옴표나 세미콜론 등의 sql 구문에서 어떤 역할을 하는 특수문자들을 모두 필터링하는 겁니다. 필터링을 할 때는 특수문자 앞에 \역슬래시를 붙여서 무력화 시킬 수 있습니다. 필터링 과정에서 정규표현식을 사용하면 편리합니다. 단 잘못된 정규표현식을 작성하는 경우, 간단한 작업임에도 서버에 부하가 걸리고 DDOS에 취약해질 수 있습니다. 그래서 저는 다음 방법을 추천합니다.

현재 사용되는 대부분의 sql에는 준비된 선언이라는 기능이 있습니다. 이 기능을 간단하게 설명하자면 sql 구문의 빠른 반복 실행을 위해 인자(매개변수)를 제외한 부분을 미리 컴파일 해둡니다. 그리고 그 후에 인자 값을 넣어서 최종 실행하는 과정을 거칩니다. 이 인자 값은 오로지 문자열 역할만 하기 때문에 sql injection을 원천적으로 예방할 수 있습니다.[11]

데이터베이스의 에러 등이 클라이언트에게 보여지지 않도록 설정하는 것도 필요합니다. 데이터베이스에서 나오는 에러 메시지 등이 injection 공격에 도움을 줄 수도 있기 때문입니다.

---

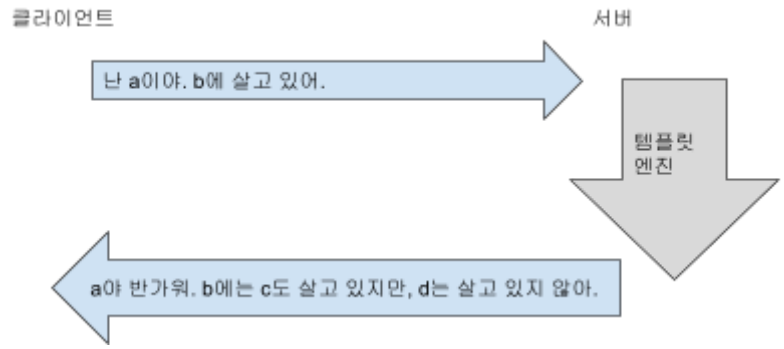
[11] 파이썬 예시
입력값 = [(1,"foo"), (2, "bar"), (3, "baz")]
cursor.executemany(" INSERT INTO data ('item_num', 'item_name') VALUES (?, ?)", 입력값)

에러: *id*라는 것은 존재하지 않습니다. *user_id*로 대신 검색해보시겠습니까*?*

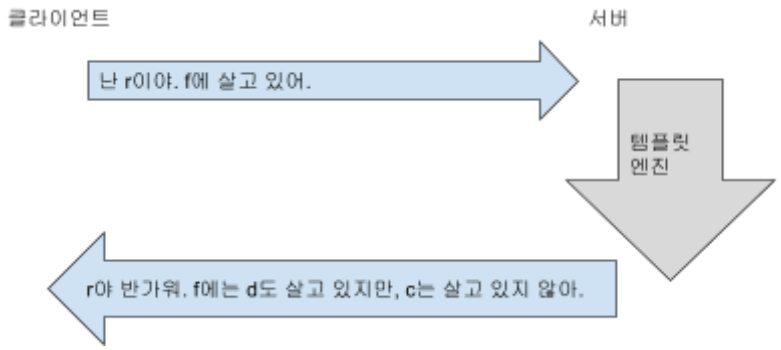
이 공격은 sql뿐만 아니라 nosql에서도 일어날 수 있고, 공격도 증가하고 있어서 nosql에도 관심을 가져야 합니다.

Server Side Template Injection:

템플릿 엔진은 미리 지정된 양식과 사용자(클라이언트)의 데이터를 잘 합쳐서 사용자에게 보여줄 최종 html을 만들어 주는 역할을 합니다.



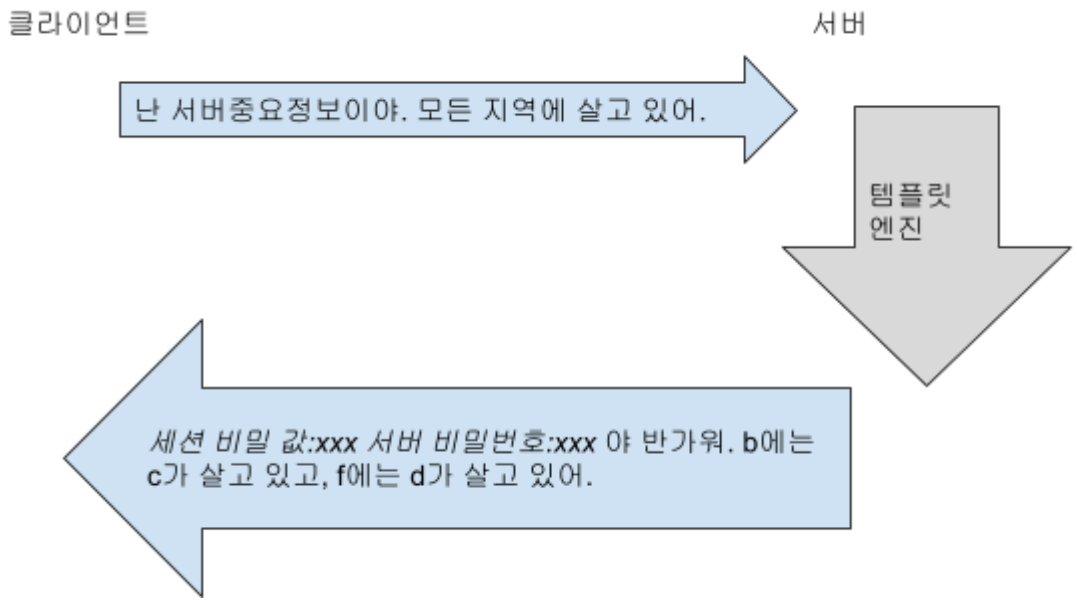
클라이언트는 자신이 a이고, b지역에 살고 있다고 보냅니다. 그럼 서버의 템플릿 엔진은 a에게 인사를 하는 부분을 만듭니다. 그리고 a가 살고 있는 b지역의 정보들을 보여줍니다. api 등을 이용해 b에 관련된 정보들을 가져와서 지정된 양식에 맞추어 잘 배치하는 겁니다.



이렇게 사용자의 입력에 따라 html을 자동적으로 바꾸고 보여줄 수 있게 되는 겁니다. 특히 똑같은 페이지에서 구체적인 데이터만 바뀌는 경우(동적 웹페이지)에는 필수입니다.

```
</head>
<body>
{% with messages = get_flashed_messages() %}
    {% if messages %}
        <script>
            alert("{{messages[-1]}}")
        </script>
    {% endif %}
{% endwith %}
<div style="border: 4px pink solid; padding: 5px">
    <h2>등록된 파일들</h2>
    <ul style="margin: 10px; padding: 10px; font-size: larger">
        {% for i in file_list %}
            <li>{{ i }}</li>
        {% endfor %}
    </ul>
</div>
{% if file_list|length >= 5 %}
    <h1>최대 5개까지 올릴 수 있습니다.</h1>
{% else %}
    <form action="/adupload" method="POST" enctype="multipart/form-data">
        <p><input type="file" onchange="checkFile(this)" name="file"></p>
        <div><input type="url" name="url" placeholder="광고 이미지를 클릭하면 url로 연결되는 방식입니다"
                    style="..."><br>광고 송출 시 같이 나갈 url을 입력하세요
        </div>
        <p><input type="submit" value="올리기" class="btns" style="..."></p>
    </form>
{% endif %}

</body>
</html>
```

이 사진은 **flask** 웹프레임워크의 기본 템플릿 엔진 **jinja2**를 이용하여 **html**을 작성한 코드입니다. 조건에 따라 어떤 **html** 엘리먼트는 보여주고, 보여주지 않고를 결정할 수도 있습니다.
그러나 잘못된 템플릿 엔진 사용은 취약점을 가져올 수 있습니다.



이렇게 클라이언트가 보내는 값이 서버의 중요 정보를 노출하도록 유도하는 값일 수도 있습니다. 이럴 경우 이에 대한 대비가 안 되어 있으면 서버의 중요 정보가 노출되고, 공격자가 쉽게 공격할 수 있는 환경이 만들어지게 되는 겁니다.

jinja2에도 이 취약점이 한정적으로 있습니다.
먼저 실습을 위해 파이썬 환경을 만들어줍니다.
https://replit.com/
https://www.jetbrains.com/ko-kr/pycharm/download/#section=windows

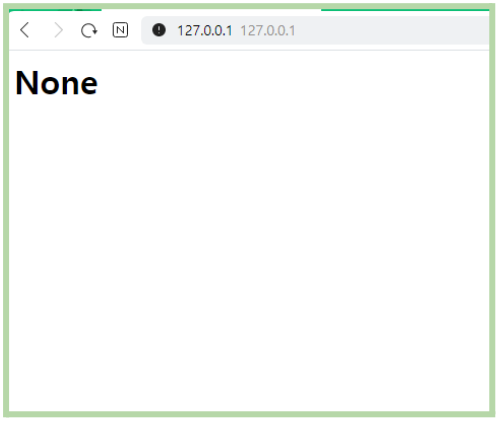
파이썬을 사용할 수 있는 환경에서 flask를 설치해줍니다.

pip install flask

app.py 파일, templates 디렉토리, static 디렉토리를 만들어 줍니다.
app.py에 다음 코드들을 넣어줍니다.

```
from flask import *
app = Flask(__name__)
app.secret_key = 'jklsad;fjkol;ewra132967-asdfp
'
@app.route('/')
def index():
    ssti = request.args.get('ssti')
    html = f'''
        <html>
            <h1>{ssti}</h1>
        </html>
    '''
    return render_template_string(html)

if __name__ == "__main__":
    app.run('0.0.0.0', port=80, debug=True)
```

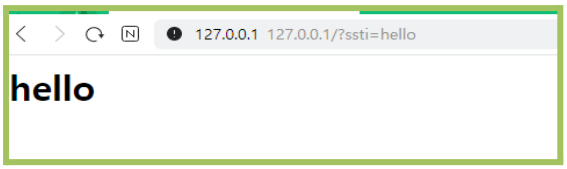
이 코드를 실행시키고 브라우저에서 localhost로 접속하면 다음과 같은 것이 나올 겁니다.



ssti라는 매개변수를 통해 인자를 입력해주어야 하는데, 아무것도 입력하지 않아서 none 값이 출력된 겁니다.

매개변수 ssti의 값을 출력해보면 None 값이 나와있는 걸 볼 수 있습니다.

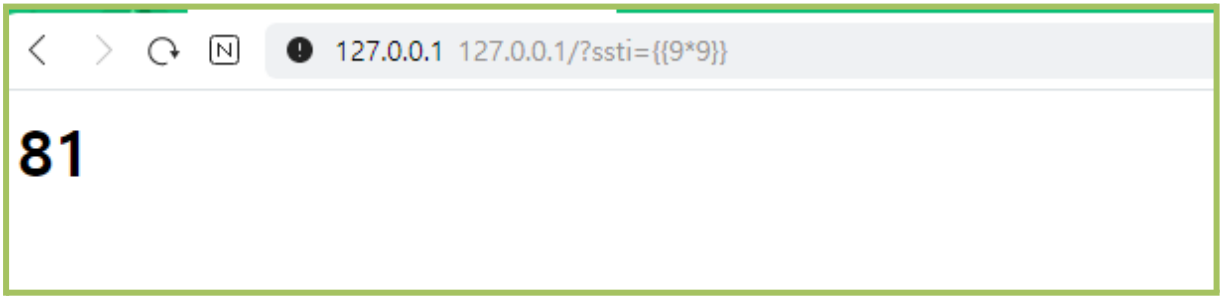
localhost/?ssti=hello를 입력하면 다음과 같이 나올 겁니다.



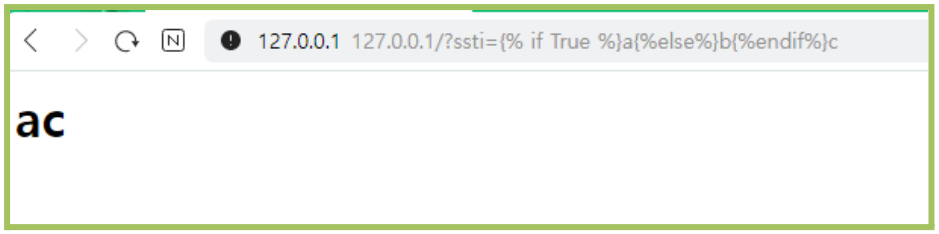
hello 대신 아무 값이나 입력해도 그대로 나올 겁니다.
jinja2는 여기서 취약점이 드러나는데, 입력한 값 중에 어떤 동적인 역할을 하는 것을 입력했을 때, 그것이 그대로 작동한다는 겁니다.
jinja2에서는 서버에서 보여주어야 하는 동적인 값을 표시할 때 {{}} 중괄호 2개로 감싸고, 그 안에 값의 이름이나 연산을 적어줍니다.
{{ 3 +3 }}이 html 안에 들어가 있으면, 사용자에게는 6이 표시된다는 겁니다.
a=hello일 때 {{a}}가 html 안에 들어가 있으면, 사용자에게는 hello가 표시됩니다.

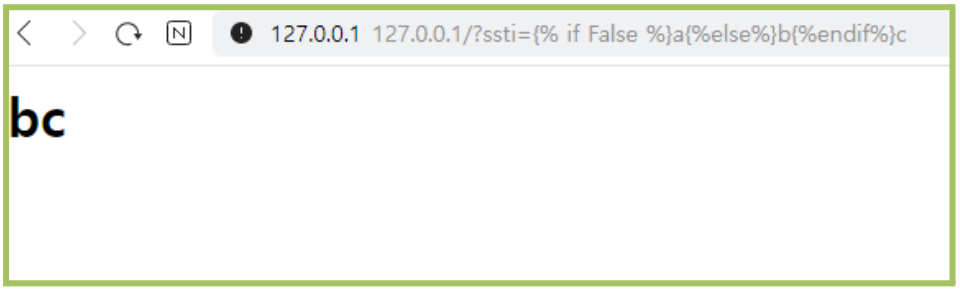
```
{% if True %}
a
{% else %}
b
{% endif %}
c
```

이런 식으로 비교 구문을 넣어줘, 사용자에게 보여줄 것을 결정할 수도 있습니다. 이 경우에서는 ac가 출력될 겁니다.
이런 jinja2 문법은 개발자만 작성할 수 있어야 하는 겁니다. 그러나 일반 클라이언트가 이 문법을 작성할 수 있게 되고, 코드 삽입 공격이 가능해집니다. localhost/?ssti={{9*9}}를 입력해보면, 입력한 것이 그대로 나오는게 아니라, 연산의 값이 나오게 됩니다.



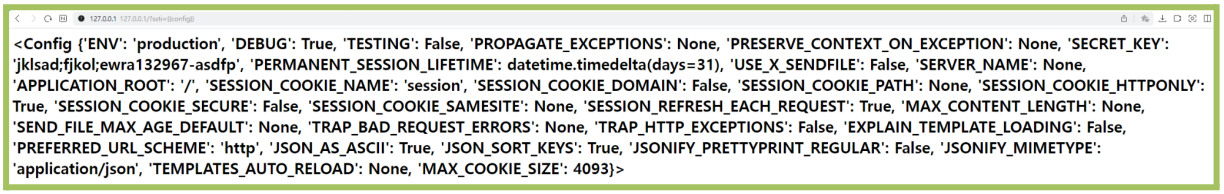
jinja2 문법이 모두 사용 가능합니다.





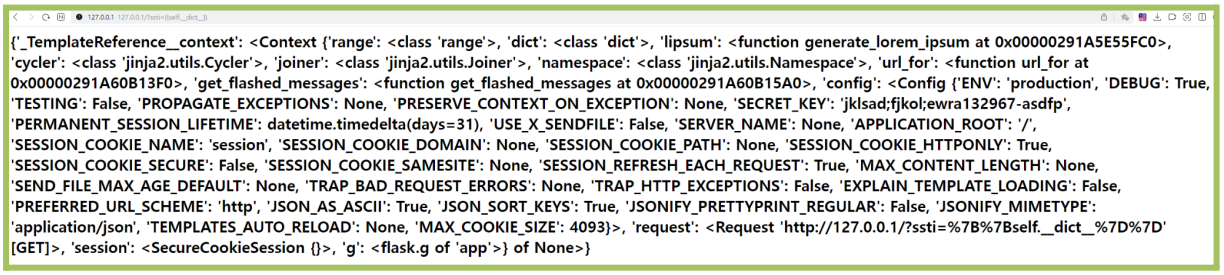
이뿐만 아니라 서버의 중요 정보도 빼올 수 있습니다.

localhost/?ssti={{config}}를 입력해서 서버 정보를 가져올 수 있습니다.[12]



시크릿 키 같은 서버 작동에서 중요한 정보들이 쉽게 클라이언트에 보여지게 되는 겁니다.

config 객체 이외에도 서버 중요 정보를 가져올 수 있는 방법은 많습니다.



이 공격의 방어를 위해서는 템플릿 엔진을 활성화시키는 특수문자(jinja2는 {{}})를 필터링하면 될 겁니다.

flask의 경우, app.py 파일이 아닌, templates 디렉토리 안에서 별도의 html을 만들고 여기에다가 jinja2를 적용할 수 있습니다. 이렇게 별도로 격리시켜놓으면 공격이 작동하지 않고, 공식 문서에도 templates 디렉토리를 이용하라고 권장하고 있습니다.[13] 이 방법을 이용하면 flask의 jinja2에 대한 ssti에 대한 방어는 완성되었다고 보면 됩니다.

---

[12] flask는 기본적으로 config 객체에 중요정보를 담아두고 있습니다.

[13] https://flask.palletsprojects.com/en/2.2.x/quickstart/#html-escaping

```
@app.route('/b')
def test():
    ssti = request.args.get('ssti')
    return render_template('b.html', title=ssti)
```

이 코드를 app.py에 추가시켜 주고, templates 디렉토리 안에 b.html을 생성합니다. html 파일 안에는 다음 코드를 추가합니다.

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <title>hello world</title>
</head>
<body>
<h1>{{title}}</h1>
</body>
</html>
```

localhost/b?ssti={{9*9}}에 접근하였을 때, jinja2가 임의로 활성화되는 일은 일어나지 않았습니다. 이 방어의 핵심은, 모든 것을 오로지 문자로만 처리한다는 겁니다. 유명 웹 프레임워크의 경우 여러 보안 패치를 통해 ssti에 대한 방어가 기본으로 제공되는 경우가 많습니다. 다만 항상 웹 개발을 할 때, ssti를 직접 테스트(템플릿 엔진 문법을 활성화 시키는 특수문자를 넣어보는 것[14])를 해보셔야 합니다.

[14] 엔진을 활성화시키는 특수문자(보통 중괄호입니다)를 넣고, 7*7 같은 수학 연산을 넣어서 테스트 해봅니다.{{7*7}} 등

만약 여기서 취약점을 발견하셨다면 반드시 엔진을 활성화시키는 특수문자를 필터링하는 코드를 넣어주어야 합니다.

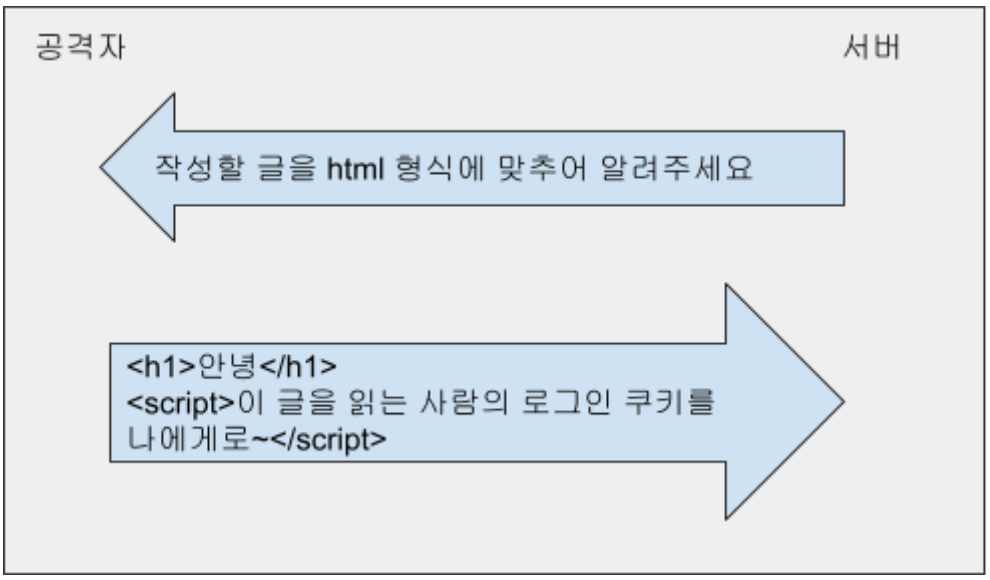
```
@app.route('/')
def index():
    ssti = request.args.get('ssti')
    ssti = ssti.replace("{{", "")
    ssti = ssti.replace("}}", "")
    html = f'''
        <html>
            <h1>{ssti}</h1>
        </html>
    '''
    return render_template_string(html)
```

다른 방법이 없고, 그대로 놔두면 보안 문제가 생긴다면 어쩔 수 없이 사용자 경험을 포기해야 할 수도 있습니다.[15]

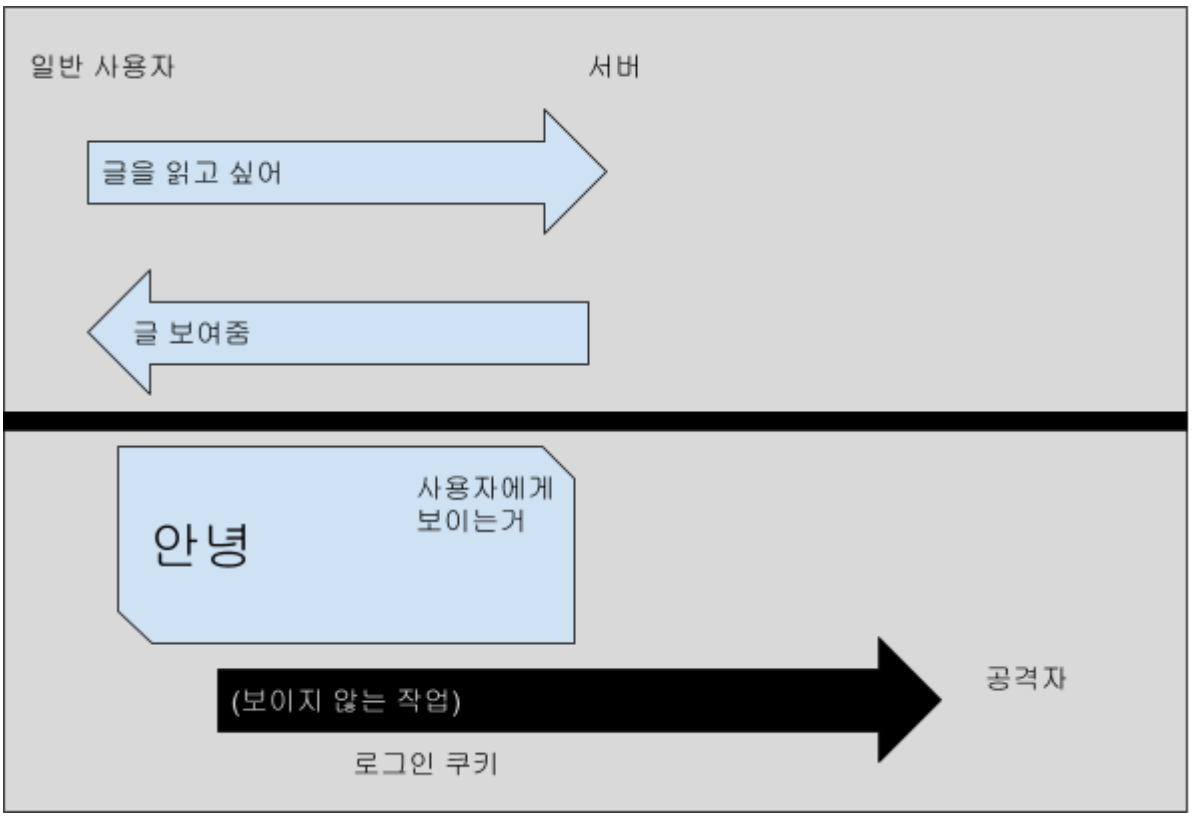
Cross Site Scripting:

줄여서 xss[16]라고 부르는 유명한 코드 삽입 공격입니다. 컴퓨터에 아무런 문제가 생기지 않았지만 xss에 당한 웹 사이트에 접속하는 것만으로 좀비pc 역할을 해주게 될 수도 있습니다.

이외에도 개인정보, 특히 로그인 정보가 강탈당하는 피해를 가져올 수도 있습니다. 즉, 단순히 웹 사이트에 접속하였는데, 공격자가 접속한 사람의 계정으로 로그인을 할 수 있게 되는 상황이 될 수 있다는 겁니다. 이 공격은 FrontEnd가 크게 관여하는 공격이고, 게시판 웹 사이트, 즉 html 텍스트 에디터 등을 이용하는 곳에서 자주 일어납니다.



[15] 이런 극단적인 상황은 일반적인 웹 개발에서 보기 어렵습니다.
[16] 원래라면 css여야 하겠지만, 동명의 다른 유명한 친구가 있어서 xss가 되었습니다.



위 상황이 이루어졌을 때, 일반 사용자는 안녕이라는 h1 크기의 글자를 보게 됩니다. 그런데 동시에 보이지 않는 곳에서 사용자의 쿠키가 공격자에게 몰래 전송되게 됩니다.
서버는 단지 공격자의 JS 코드를 옮겨주는 역할만 하게 됩니다. 진짜 목적은 사용자인 것이죠.

단순히 html을 직접 입력 받는 것뿐만 아니라 인프런 블로그처럼 gui 에디터로 글을 작성하긴 하지만, 사실은 뒤에서 html 소스를 작성해나가고 있는 곳에서 공격이 많이 이루어집니다.[17] 즉, 이것도 마찬가지로 사용자의 입력을 그대로 다시 사용자에게 보여주는 패턴에서 공격이 발생하는 흔한 유형이였던 것이죠.

[17] 인프런 블로그는 제가 테스트해본 결과 xss에서 안전하더라고요. 조금있다가 나올 고급 기법 xss도 모두 backend에서 필터링 되었습니다. 다행인 것 같습니다.

웹 사이트에서 취약점이 확인된다면 자바스크립트 코드를 아무거나 넣을 수 있기 때문에 무궁무진하게 활용할 수 있습니다.[18] 이 공격이 위협적인 또다른 이유는 일반 사용자 입장에서는 공격자가 삽입한 자바스크립트랑 원래 html에 있던 자바스크립트와 구분이 안 된다는 겁니다. html에 들어있는 자바스크립트는 다 똑같이 취급 되니까요. 웹 개발에 능숙한 사람이 페이지 소스를 보고 찾아내기 전까지는 일반 사용자는 공격이 이루어지는 지 알 수 없습니다.

간단하지만 치명적인 공격인데, 방어하기도 쉽지 않습니다.
단순히 script 태그를 차단하면 될까요? script 태그 없이도 JS를 실행시킬 수 있습니다.

```
<a href="javascript:alert('hello,world')">눌러보세요</a>
```

이처럼 사용자에게 클릭을 유도하고, 스크립트를 실행시키게 할 수 있습니다.
그럼 javascript라는 단어를 차단하면 될까요?

```
<a
href="&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3A;&#x61;&#x6C;&#x65;&#x72;&#x74;&#x28;&#x27;&#x68;&#x65;&#x6C;&#x6C;&#x6F;&#x2C;&#x77;&#x6F;&#x72;&#x6C;&#x64;&#x27;&#x29;">눌러보세요</a>
```

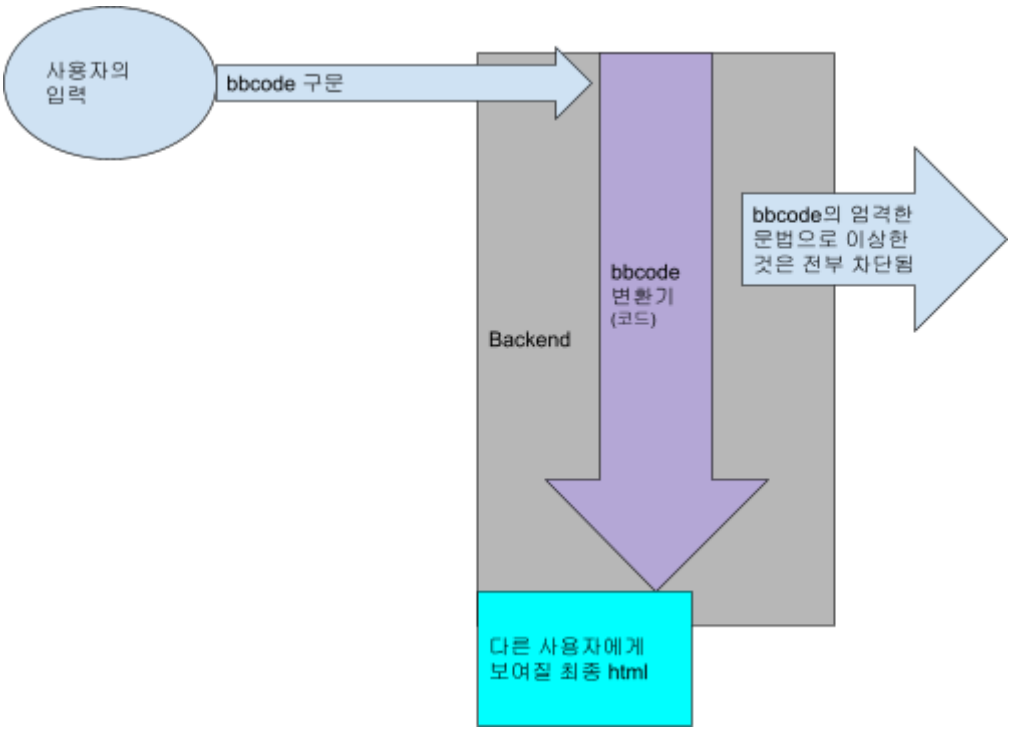
각 문자를 난독화해서 필터링을 피할 수 있습니다.

html 엘리먼트를 사용하는 것이 아닌, 오직 텍스트만 사용하는 곳에서는 이런 문제가 생길 가능성이 원천적으로 없습니다. 그러나 텍스트만 있으면 밋밋하기 때문에 사용자 경험을 위해서라면 어쩔 수 없이 이 방법은 후순위로 고려될 겁니다.

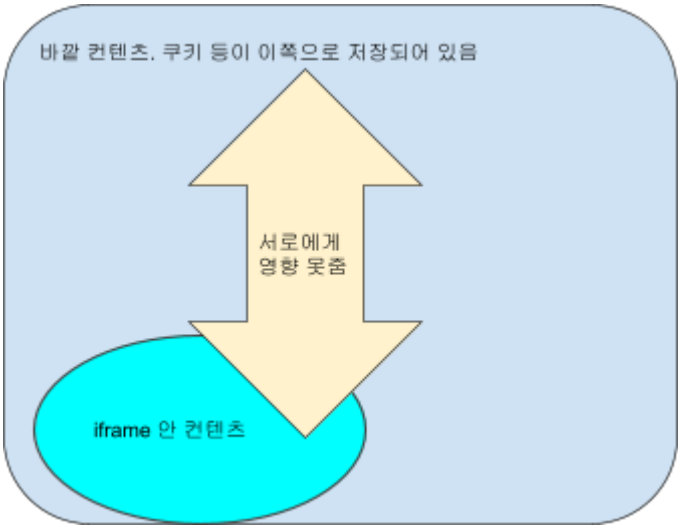
현재 다양한 곳에서 쓰이며, xss에서 안전한 방법은 BBcode를 사용하는 겁니다. bbcode는 간단한 마크업 언어로, 위키 등에서 많이 사용됩니다. 결국은 변환 과정을 거쳐 html을 보여주는 것이지만, 사용자의 입력을 그대로 다시 보여주는 것이 아니라서, 안전한 방법입니다. 특히 bbcode는 악성 JS를 삽입할 가능성을 봉쇄(스크립트 관련 문법이 존재하지 않음+엄격한 문법)하였기 때문에, 저는 BBcode 사용을 추천드립니다.
bbcode 문법 중에 링크를 연결하는 (html에서는 href 속성) 부분에는 http나 www로 문자열이 시작해야 합니다. 이 문법을 강제하지 않는 라이브러리더라도, 변환과정에서 자동적으로 http://를 붙여줄 수 있습니다. xss에서 안전하면서, 또 사용자가 편하게 gui 텍스트에디터로 사용할 수 있게 해주는 라이브러리도 있습니다.[19]

---

[18] https://twitter.com/dergeruhn/status/476764918763749376
지금은 평범한 트윗으로 보이겠지만, 링크에 접속하는 것만으로 자동적으로 리트윗 버튼이 눌러지게 되는 시절이 있었습니다.
[19] 사용자의 gui 입력을 bbcode 문법 형식으로 변환하고, 이를 또 backend가 다시 보여줄 때 html로 변환하는 겁니다.



텍스트 에디터를 웹 사이트에 적용할 때는 앞서 나온 것처럼 bbcode를 사용하는 텍스트 에디터를 추천하지만, 단순 html 텍스트 에디터를 이용할 때도 xss 방어 로직을 넣을 수 있습니다. 기본적으로 backend에서 입력으로 들어온 것들 중에서 script 태그과 그 안에 있는 것들은 제거합니다. 그리고 a 태그의 href 속성 등에는 http(s)나 www로 시작하는 내용 등만 받습니다. 그외 나머지 경우는 모두 제거하는 로직을 구성합니다. html 에디터를 적용하시겠다면 굳이 말릴 생각은 아닙니다. 단, 보안을 위해 꼼꼼하게 입력 값 검사를 진행하여야 한다는 것을 명심하셔야 됩니다.[20]



이외에도 콘텐츠를 iframe 태그 안에 가둬버리는 방법도 있습니다. iframe 태그 안에 생기는 html과 원래 html은 서로 독립적인 성질(SOP)을 이용하는 겁니다. 즉, 공격자가 xss 공격을 해도, iframe 밖으로 공격이 나가지 않아서, 쿠키 등이 저장되어 있는 바깥에는 영향을 주지 않는 원리를 이용하는 겁니다.
물론 이 방법은, 바깥 콘텐츠에 접근하는 것을 제외한 모든 것이 가능해집니다. 꼭, 이점을 명심하고, 이 방법을 사용하셔야 합니다.

[20] 인프런처럼요!

X-XSS-Protection라는 http 헤더 속성을 넣을 수도 있습니다. 헤더 값을 1로 넣어주면, 브라우저에서 xss 공격이라 판단한 것을 막아줍니다. 또한 httponly 쿠키 설정을 통해, 자바스크립트가 접근하지 못하도록 막아줄 수 있습니다. 다만 매우 일반적인 공격에 한해서 막아줄 수 있고, 능숙한 공격자의 공격은 못 막아줄 가능성이 큽니다. 이것만 사용하는 것은 위험하지만, 다른 대책과 함께 사용하는 것은 나쁘지 않습니다.

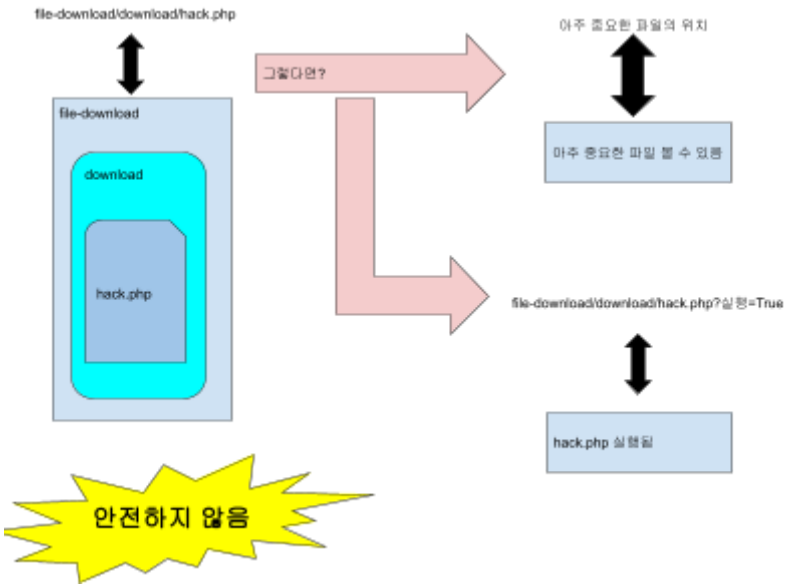
파일 업로드 취약점:
다른 Injection 공격과는 거리가 있지만, 이 공격도 마찬가지로 의도하지 않은 어떤 것을 삽입해서 공격하는 것이라서 넣었습니다.
파일을 업로드하는 웹 서비스(클라우드, 게시판)에서 운영체제에 영향을 줄 수 있는 파일을 업로드하고 실행하여, 공격하는 유형입니다. 이 파일에는 웹에서 운영체제 명령어를 칠 수 있는 웹쉘, 악성코드 파일, 더 나아가 서버를 실시간으로 감청하는 실행파일 등 다양합니다. 간단하지만 많이 당하는 공격입니다.
이 공격을 예방하기 위해, 특정 확장자(exe, php 등)를 제한하도록 할 수 있습니다. 가장 일반적이고 편리한 방법입니다. 그러나 이건 순수하게 이용하려던 사용자 경험에 좋지 못한 영향을 줄 수 있다는 것을 명심해야 합니다.
만약 사진만 올리는 곳이라면, 이미지 확장자를 제외한 다른 확장자는 frontend, backend에서 모두 안된다고 막아두는게 좋습니다. 이렇게 하면 공격도 막으면서, 사진 파일을 잘못올린 사용자의 사용자 경험도 개선할 수 있습니다.

이 공격을 잘 방어하는 핵심은 격리입니다.[21] 공격자가 파일을 업로드 한 후, 이 파일에는 직접 접근할 수 없게 해야 한다는 겁니다.

지금 나올 공격의 예시는 안전하지 않은 웹 애플리케이션의 경우입니다. 공격자는 서버에 올려진 파일을 다시 다운로드 하는 과정에서 파일이 저장된 위치를 알아낼 수 있습니다.
www.inflearn.com/file-download/download/hack.php
다운로드 소스가 이런 식으로 되어있다면, file-download/download 디렉토리에 hack.php가 위치한다는 것을 알아낼 수 있습니다. 공격자는 공격 파일이 저장된 디렉토리 위치를 찾고, 이 파일을 실행시킵니다. 파일 실행에는 get 방식으로 파라미터를 보내준 것에 의해 실행될 수 있습니다.
이렇게 url의 path가 운영체제에 직접 영향을 미치는 방식은 정말 위험한 방식이며, 업로드 및 다운로드 링크로 사용자가 디렉토리 구조를 알 수 없게 해야 합니다. 이런 식의 코드는 굳이 이 공격이 아니더라도 큰 피해를 입힐 수 있습니다.*www.inflearn.com/../../../../etc/passwd*

[21] 밑에서는 소프트웨어적 격리만을 이야기 하였지만, 물리적 격리도 나쁘지 않습니다. object storage 같은 것도 이 공격을 방어하는데 큰 도움이 됩니다.

그리고 path에서 불필요한 get 파라미터를 제거해주고 요청을 처리하는게 좋습니다.
가장 중요한 것은, URL의 path, 매개변수가 운영체제에 직접 접근, 영향을 끼치게 해서는 안 된다는 겁니다. url의 path는 단지 웹 애플리케이션이 어떤 일을 해야 할지 지정해주는 것으로만 제한해야 합니다. [22]



안전하게 구성된 웹 애플리케이션에서 file-download/download의 역할은, backend가 어딘가에 저장되어 있는 hack.php를 frontend로 전송해주는 로직(함수, 라우트 등)만 수행하는 겁니다. url path로 직접 파일에 접근하는 것이 아닌 것이죠.

이것 외에도, 사용자에게 받은 파일을 그대로 저장해서는 안됩니다. 안전하게 입력받은 파일을 격리하였더라도, 파일 이름으로 인해 운영체제가 공격 받을 수 있습니다. 파일 이름을 그대로 저장하는 것은 추천하지 않고, Hash 테이블을 사용하는 것을 추천합니다. 귀찮아도 별도의 db를 만들고 이런 식으로 데이터를 넣는 것이 안전에 큰 도움이 됩니다.

파일을 올리는 곳에 저장된 이름	사용자가 올린 파일의 실제 이름
abcd.php	../../../.php
efgh.exe	!@#$%^&*().exe
ijkl.shell	hello.shell

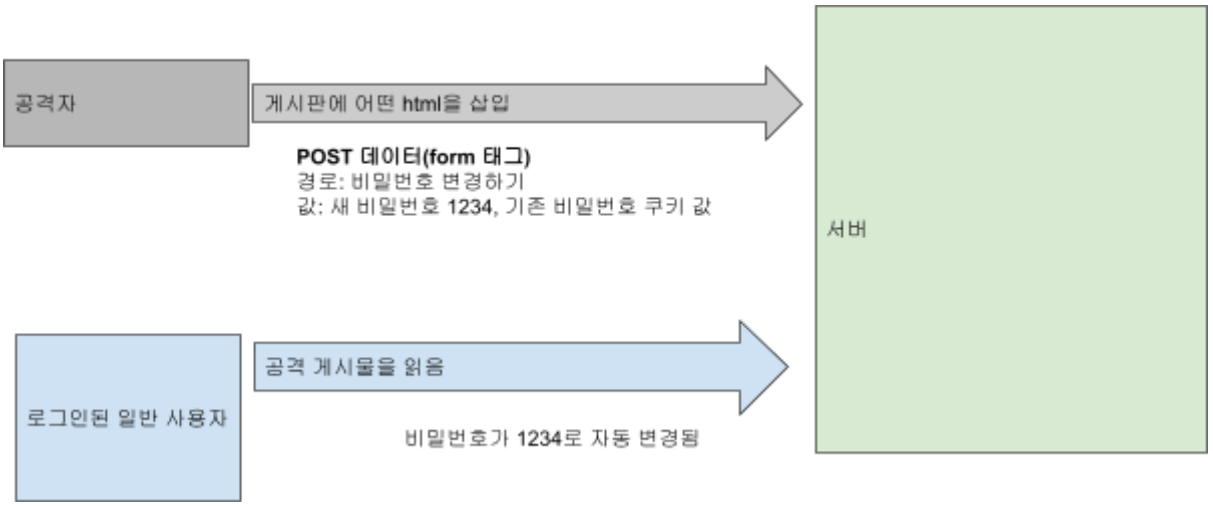
단지 업로드, 다운로드 역할만 해줄 것이라면 파일이 손상되지 않는 선에서 아예 확장자도 빼버리고 저장해도 좋습니다. 업로드할 때는 아무것도 못하는 데이터로 올려졌다가, 다운로드를 할 때는 DB에 맞추어 다시 이름과 확장자를 원래대로 되돌리고, 사용자에게 파일을 주는 것입니다.

[22] 이게 안된다면 어쩔 수 없이 입력된 파일에 대해 하나하나 검사를 해야될 수 밖에 없습니다. 이를 테스트하는 방법은 url로 어떤 위치의 텍스트 파일을 열어보는 것으로 알아낼 수 있습니다.

참고로 파일의 확장자를 찾을 때는, 단순히 이름에 있는 확장자가 아닌, 파일 헤더에 명시된 실제 확장자를 이용해야 합니다.

CSRF:

XSS와 비슷합니다. 너무 뻔한 공격이라고 생각하지만, 매우 많이 일어나는 공격이며 대응도 까다롭습니다.



정상적으로 접속을 하고 인증을 한 사용자에서 보낸 요청에(비록 그것이 사용자가 원치않는 요청이더라도) 대해 서버가 신뢰하고, 어떤 로직을 수행한다는 것을 이용한 공격입니다.
이 공격은 간단하게 어떤 요청을 수행하는 경로, 데이터로 보내줘야 하는 값만 정확하게 파악하고 있으면 form 태그 등을 삽입하여 쉽게 할 수 있습니다. 사용자는 공격자가 요구한 클릭 한 번(form submit)으로 엄청 큰 피해가 일어날 수 있는 겁니다.[23] 직접 공격자에게 데이터를 전송하는 것 등은 아니지만, 위 그림에 나온 것처럼 비밀번호 변경 로직을 이용해 공격자가 html을 읽은 로그인된 사용자의 비밀번호를 임의로 변경하는 등, 큰 취약점을 생성해낼 수 있습니다.

사용자가 원하지 않는 요청을 보내지 않기 위해 캡챠 등을 사용하여 방어할 수 있습니다. 캡챠가 가장 안전한 방어 방법이지만, 다른 방법들도 있습니다.
먼저 랜덤한 값의 세션을 생성합니다. backend에서 사용자에게 html을 보여주기 전에 템플릿 엔진 등으로 세션에 연결된 사용자에게 세션 관련 값을 넣어줍니다. 보통 input hidden으로 넣어줍니다. 사용자는 어떤 요청을 보낼 때, 이 세션 관련 값을 함께 다시 보내는 겁니다. 서버는 저장된 세션 값과 사용자가 보낸 이 값을 비교하여, 요청을 신뢰할 수 있게 되는 겁니다.
만약 공격자가 어떤 요청을 자동적으로 보낼려고 한다면, 이 랜덤한 세션 값을 알 수 없기 때문에 공격을 할 수 없게 됩니다.

---

[23] 아마 여기서 XSS가 연상이 되실 겁니다. 만약 xss 공격이 합쳐졌다면, form을 자동으로 보내는 스크립트 등이 추가로 삽입되었을 겁니다.

많은 웹 프레임워크가 이 csrf 방어를 기본으로 내장하고 있습니다. 사용하시는 웹 프레임워크가 있다면, 어떤 식으로 csrf 방어를 하는 지 알아보시는게 도움이 될 겁니다.

- 무차별 대입

영어로는 brute force라고 부릅니다. 이 공격은 암호랑 큰 관련이 있습니다. 조합 가능한 모든 문자열을 전부 조합하여 암호를 해독하는 겁니다. 4칸짜리 자물쇠가 있었을 때, 0000부터 9999까지 하나씩 다 해보는 겁니다.

이 공격이 무서운 점이 암호를 해독할 정확도가 100%라는 겁니다.
그러나 이 공격의 약점은, 암호를 해독하기 위해 하나씩 대입하는데 천문학적인 자원과 시간이 들어갈 수 있다는 겁니다.

간단하다고 무시하면 안되는게, 뒤에 암호부분에 DES라는 암호가 나옵니다. 이 암호의 구조는 견고해서 지금 기준으로 나쁘지 않지만, 심각한 문제는 무차별 대입에 약하다는 겁니다. 사소한 문제가 있긴 하지만 최신 고사양 컴퓨터들에 의해 이루어지는 무차별 대입이 주원인으로 현재는 사용이 비권장되는 암호입니다.

사전 공격 같은 발전형 공격도 존재합니다. 특히 로그인 비밀번호에 많이 쓰이는 공격으로, 미리 사람들이 많이 사용하는 비밀번호를 모아둡니다. 그리고 우선적으로 이 목록을 넣어서 뚫어보는 겁니다.

https://haveibeenpwned.com/Passwords

이 사이트에서 비밀번호를 입력하면 공격자들이 많이 사용하는 사전에 포함이 되어 있는지 확인할 수 있습니다.[24]

누구든 상관없이, 비밀번호는 공격자가 쉽게 알 수 없게 만드는 게 중요합니다. 웹 개발자가 열심히 보안 조치를 해봤자, 너무 쉬운 비밀번호는 웹 사이트의 보안 조치를 아무 의미 없게 만들어 버립니다.

[24] 정확히는 침해된 적이 있는 비밀번호를 조회하는 사이트입니다. 이전에 자주 유출된 비밀번호라면 아마 사전에 들어가 있을 가능성이 매우 큽니다.

웹 개발자는 사용자가 회원가입을 할 때 너무 쉬운 비밀번호는 설정하지 못하도록 해야 합니다. 또한 너무 비정상적인 로그인 요청 등을 잘 파악해야 합니다.

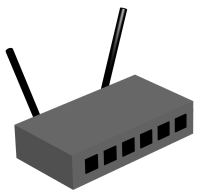
저는 예전에 어떤 사이트의 보안 테스트를 진행해본 적 있습니다.[25] 여기에다가 비정상적인 요청을 보내니, 어떤 http 쿠키가 할당되었다는 것을 확인하였습니다. 계속 요청을 보내니 쿠키의 값이 변경되어가는 것을 확인했습니다. 아마 값의 크기가 증가되어간다는 의미겠죠? 어느 순간이 되었을 때, 요청을 보내니, 이제 시도가 불가능하다는 메시지와 함께, 요청을 보낼 수 없게 되었습니다. 그래서 저는 요청을 지속적으로 보낼 때, 응답으로 돌아오는 쿠키를 자동적으로 삭제할 수 있도록 조치를 해두고 계속 공격 시도를 하였답니다.

비정상적인 사용자를 차단할 때는 위에 나온 저의 사례처럼 너무 단순한 방식은 사용하지 말아야 합니다. 쿠키처럼 frontend쪽에서 임의로 수정할 수 있는 값을 사용하는 것은 공격자를 제대로 식별해내기 매우 어렵습니다.
금융 사이트를 비롯한 여러 웹 서비스에서는 전화번호 인증 등을 해야 다시 요청을 보낼 수 있게 해놓았습니다. 다중 인증은 사용자 입장에서는 불편해도 정말 안전합니다. 작은 웹 사이트에서는 캡챠를 많이 이용합니다. 캡챠도 잘 이용하면 충분히 안전합니다.

단, 이런 조치를 취할 때는 조심해야 할 부분이 있습니다. 단순히 이런 유용한 도구들을 frontend에서 접근을 제한하는데만 사용해서는 안됩니다. 브라우저 화면에는 접근이 제한되었다고 나오지만, 직접 post로 데이터를 보내보니 멀쩡히 동작하는 경우가 잘못된 사용이라는 겁니다.

- 도청(스니핑)

모스부호로 통신을 할 때부터 있었던 아주 오래된 보안 위협입니다. 단순히 다른 사람의 대화를 엿듣는 것까지 스니핑으로 포함한다면, 이 공격은 끝이 없을 겁니다.
일단은 인터넷 세계에서 일어나는 도청에 집중해보도록 하겠습니다. 이 공격은 기본적으로 기밀성을 침해합니다.



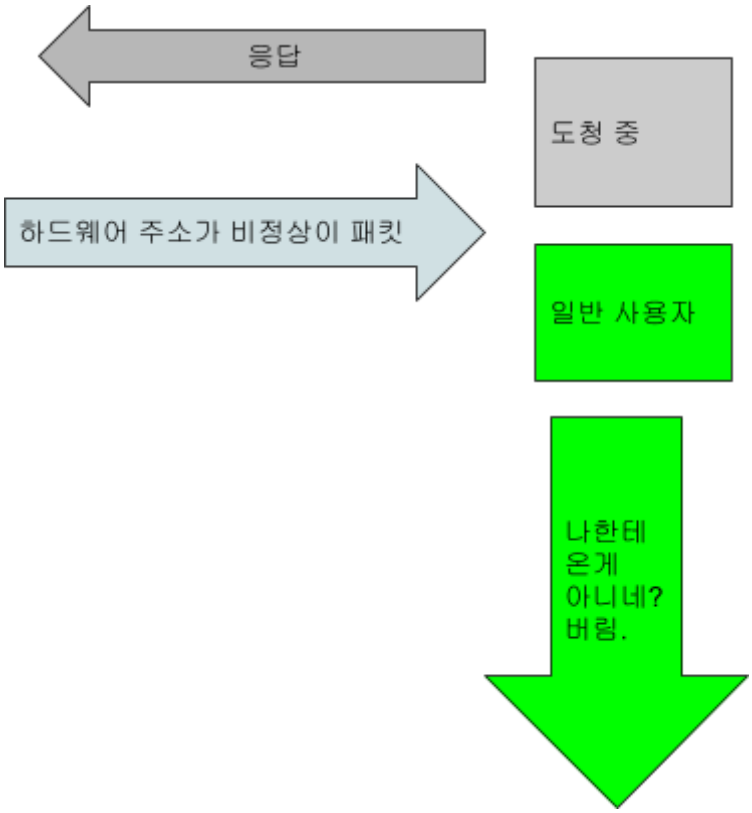
네트워크와 연결되어 있는 컴퓨터는 자신에게 오는 네트워크 패킷만 오는 것이 아닙니다. 컴퓨터를 네트워크 허브에 연결하는 경우에는 같은 허브에 연결된 다른 컴퓨터에게 가야할 패킷도 같이 들어옵니다. 앞서 나온 브로드캐스팅입니다.

[25] 물론 무보수로요.

일반적인 경우라면, 자신이 목적지가 아닌 패킷은 버립니다. 그러나 컴퓨터에 특정한 설정을 통해 들어오는 모든 패킷을 다 받아볼 수 있습니다. 단, 같은 허브에 연결된 대상만 도청할 수 있습니다.

이 공격에 대한 즉각적인 대처는 어렵습니다. 도청을 당하고 있는 쪽에서 눈치를 채기 전까지 공격이 일어났다는 것을 알 수가 없습니다. 능동적인 탐지로 공격을 방어해야 합니다.

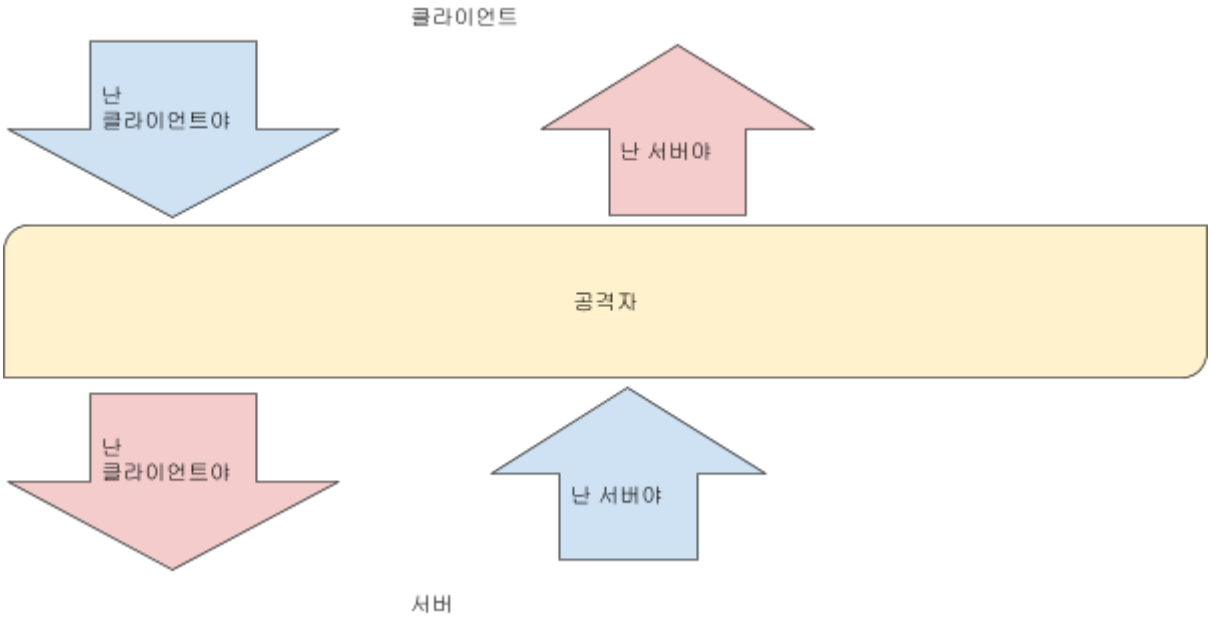
자주 쓰이는 탐지 방법은, 의심되는 시스템에 icmp 패킷을 보냅니다. 여기서 목적지의 ip는 의심되는 시스템을 의미하지만, 하드웨어 주소는 존재하지 않는 주소로 해서 보냅니다. 정상적인 시스템이라면 하드웨어 주소가 달라서, 패킷이 걸러집니다. 근데 만약 모든 패킷을 다 받도록 허용한 경우라면 이 패킷을 받을 것이고, 응답을 보낼 겁니다. 응답을 보낸 시스템이 스니핑을 시도 중이라는 것을 알 수 있습니다.



중요한 점은, 아무리 도청을 당해도 무슨 내용이 오고가는지 모르게 할 수 있다는 겁니다.
흔히 보이는 https 연결을 사용하면, 통신과정에서 송수신되는 모든 패킷을 뜯어보아도 무슨 내용인지 알 수 없게 됩니다.
자세한 내용은 뒤에 암호부분에서 나오지만, 안전한 암호 시스템을 이용하는 게 아주 큰 도움이 됩니다.

- 스푸핑

스푸핑의 핵심은 사칭이라고 생각해주시면 됩니다.
다양한 스푸핑 공격이 있지만, 기본적으로 다음의 구조를 가지고 있습니다.

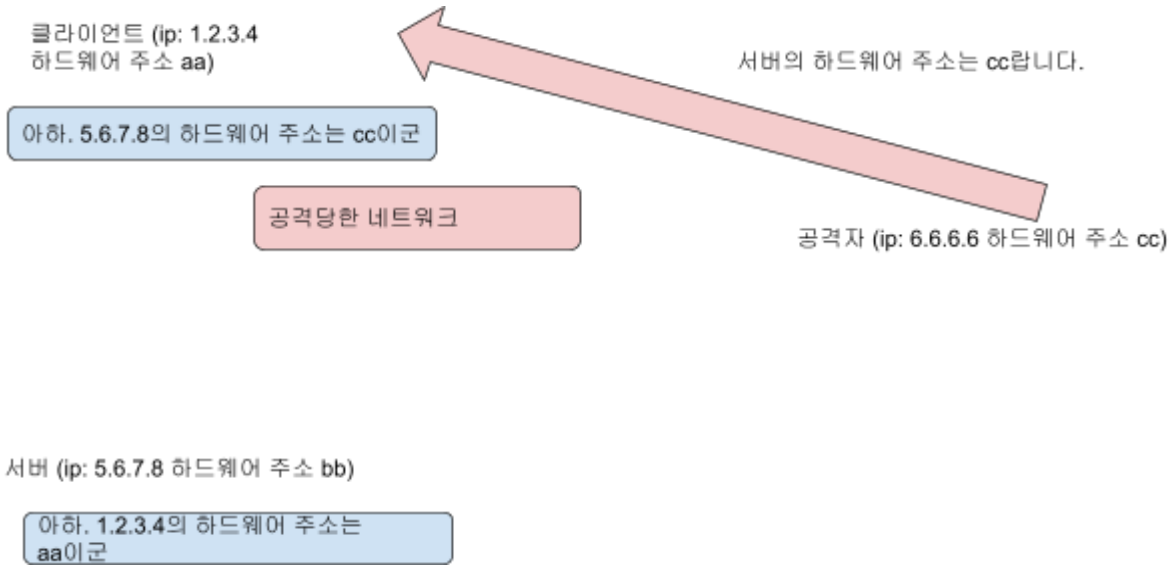


붉은 색 부분이 공격자가 사칭을 하는 부분입니다. 이 방법을 쓰면, 도청은 물론, 보낸 값을 변경할 수 있습니다. 또 이를 이용해 DOS를 할 수도 있습니다.

스푸핑에도 여러 종류가 있습니다만 근본적으로 거의 똑같아서, arp를 이용하는 스푸핑만 살펴보도록 하겠습니다.

시스템간 통신을 하기 위해서는 ip 주소와 물리적 주소(mac)가 모두 필요합니다.[26] ip 주소만 알고 있을 때, 하드웨어 주소를 알아내기 위해 사용하는 프로토콜이 ARP입니다. 시스템간 거리에 따라 ip로 mac 주소를 얻는 방식이 다르지만, 공통적으로 도착지의 네트워크에서 브로드캐스팅을 한다는 겁니다. 목표 시스템이 있는 네트워크에서, 출발 시스템이 보낸 패킷을 브로드캐스트하여 보냅니다. 패킷을 받은 컴퓨터들은 목표 ip가 본인의 ip인지 확인합니다. 만약 같다면, 패킷에 적혀있는 출발지 ip와 하드웨어 주소를 가지고 자신의 하드웨어 주소를 바로 응답을 보내줍니다. 패킷을 보낸 출발지에서는 이 정보를 일정시간 동안 저장해두고, 그쪽으로 다시 정보를 보낼 때, 이를 이용하는 겁니다. 새로운 ip의 하드웨어 주소를 찾거나, 저장 시간이 만료된 경우, 이 과정을 계속 반복합니다.
같은 네트워크에 숨어 있는 공격자는 두 시스템이 통신을 시작하려는 것을 알아냅니다. 그 후 같은 네트워크에 있는 시스템에 계속해서, 통신하려는 대상의 ip에 대한 하드웨어 주소가 공격자의 하드웨어 주소라고 속여서 보냅니다. 보통 ip에 대한 하드웨어 주소를 저장해두는 것이 동적으로 작동하여 새 응답이 들어오면, 아무 의심 없이 값을 업데이트 합니다.

[26] ip주소만 있으면 통신할 수 있다는 건, 흔한 오해죠.



간단하게 두 시스템간 통신을 완전히 장악할 수 있게 됩니다.

이 공격은 네트워크 구조상 한계점을 이용하기 때문에, 방어가 어렵습니다. 그러나, 아무리 스푸핑을 당해도, 안전하게 데이터를 지킬 수 있는 방법은 있습니다.
기본적으로 암호화가 적용되어 있으면, 패킷을 아무리 봐도 내용을 알 수 없습니다. 그러나 암호화만 적용하였다면, 사칭을 해서 내용을 보내는 것을 찾아낼 수 없을 겁니다. 다행히 현대의 암호 통신은 내가 누구인지 증명할 수 있는 방법도 적용합니다. 이것까지 이용하면 사칭하는 공격자가 고의적으로 내용을 변경하여 보내는 것까지 잡아낼 수 있게 됩니다.

물론 그렇다고 arp 스푸핑을 가만히 놔두어서는 안됩니다. arp 스푸핑과 다른 공격을 연계하면, 실질적으로 현대의 통신 보안까지 무력화 시킬 여지가 있습니다.
비정상적인 arp를 차단하는 서비스나 arp를 자동적으로 모니터링해주는 서비스가 많이 나와있고 적용되어 있습니다. 웹 서버 등 컴퓨터에서 평소보다 네트워크 지연이 발생한다면, 공격을 의심해봐야 합니다.

- SOP와 CORS

이건 보안 공격이 아니지만, 보안과 관련해서 정말 중요해서 넣게 되었습니다. 먼저 sop는 동일 출처 정책을 의미합니다. 어떠한 웹 페이지가 다른 출처(url)에서 가져온 것과 상호작용을 하지 못하게 금지하는 정책입니다. 많은 브라우저가 이 정책을 적용하고 있기 때문에 frontend 개발에 크게 영향을 끼치는 정책입니다. 이건 웹 페이지간 상호작용을 차단하기 때문에, 단지 CSS, JS, img 파일 등을 불러오는 것은 영향을 주지 않습니다. iframe 등으로 웹 페이지를 직접 가져왔을 때, 두 웹 페이지가 소통을 하지 못하게 막는 겁니다.

여기서 의미하는 다른 출처는 다음과 같습니다.
http://hello.com을 기준으로 하였을 때 입니다.

1. http://world.com(아예 다른 도메인)
2. https://hello.com(프로토콜이 다른 도메인)
3. http://mobile.hello.com(서브 도메인이 다른 도메인)
4. http://hello.com:5000(포트가 다른 도메인)

이렇게 매우 엄격하게 적용됩니다.
이 정책을 사용하는 이유는 오로지 보안 때문입니다. sop가 없을 때 보안 문제를 보시면, xss와 매우 유사할 겁니다.

1. https://world.com에 들어가면, 브라우저의 로그인 쿠키로 자동 로그인이 돼서 사용자의 민감한 개인정보를 볼 수 있다고 해보자.
2. 이때 공격자의 웹사이트 https://hello.com은 사용자의 클릭을 유도한다.
3. 사용자가 https://hello.com에 접속할 경우, 프론트엔드에서 world.com의 정보를 불러온다. (iframe 태그 등으로)
4. 그리고 world.com에 있는 내용을 hello.com의 서버로 보낸다.
5. hello.com의 서버는 자연스럽게, 사용자의 민감한 정보를 빼왔다.

4번 과정에서는 보통 자바스크립트 DOM을 이용할 겁니다. SOP 정책이 적용된 현재에는 4번 과정이 아예 안될 겁니다. ajax를 사용하면 3번에서 막힙니다. iframe 뿐만 아니라, XMLHttpRequest 등을 이용한 ajax 등, 다른 웹 페이지에 접근할 여지가 있으면, Sop에 의해 전부 차단됩니다.

xss의 경우, 앞서 나왔던 것처럼 world.com에 직접 자바스크립트를 넣어서, 공격자에게 보내는 방식이라서, sop에 위배되지는 않습니다.

근데, rest api 등을 이용할 때는, 가끔 다른 도메인으로 데이터를 받아와야 할 때가 있습니다. 그때 사용하는 것이 CORS입니다. http헤더를 추가하여 교차출처리소스공유(CORS)를 허용해주어야 합니다. 몇몇 꼼수를 이용해 제한적으로

sop를 무시할 수 있지만, 실제 배포할 프로젝트에서는 공식적인 방법으로 CORS 허용을 하실 길 추천드립니다.

이 http헤더를 api 서버 등 콘텐츠를 보내는 서버가 가지고 있으면 됩니다.
Access-Control-Allow-Origin: http://hello.com
이 http헤더를 world.com이 가지고 있을 경우, hello.com에서 웹 페이지에 대한 접근이 가능해집니다. 자신에게 접근할 수 있는 웹 페이지의 명단을 이 헤더의 값에 적으면 됩니다.
모든 곳에서 자신에게 접근할 수 있게 하려면 *을 넣으면 됩니다.
콘텐츠를 가져올 수 있게 허락해주는 것은, 콘텐츠 주인이 결정합니다.

많은 웹 프레임워크에서도 CORS 허용 헤더를 추가할 수 있게 제공해주고 있습니다.

flask의 경우 pip install flask-cors로 추가 패키지를 설치하여 사용할 수 있습니다.
정보를 요청하는 쪽

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <title>cors</title>
</head>
<body>
<input type="button" value="cors 테스트" id="btn">
<script>
    let btns = document.getElementById("btn");
    btns.addEventListener("click", () => {
        let xhr = new XMLHttpRequest()
        xhr.open('GET', 'http://127.0.0.1/b?ssti=hello', true)
        xhr.send()
        xhr.onload = () => {
            if (xhr.status == 200)
            {
                console.log(xhr.response);
                console.log("통신 성공")
            }
            else {
                console.log("통신 실패")
            }
        }
    })
</script>
```

```
</body>
</html>
```

정보를 제공해주는 쪽

```
from flask import *
from flask_cors import *
import os
app = Flask(__name__)
app.secret_key = 'jklsad;fjkol;ewra132967-asdfp'
CORS(app, resources={r'*': {'origins': 'http://localhost:63342'}})

@app.route('/b')
def test():
    ssti = request.args.get('ssti')
    return render_template('b.html', title=ssti)

if __name__ == "__main__":
    app.run('0.0.0.0', port=80, debug=True)
```

노란 부분이 flask에서 cors 허용을 해주는 부분입니다. 이전에 사용하였던 코드에서 cors 부분만 추가로 들어간 것입니다. 참고로 정보를 요청하는 쪽의 주소는 localhost:63342 입니다.[27] html에서 버튼을 클릭하고, 콘솔을 보면 b.html이 그대로 적혀있을 것습니다. 노란색 부분을 지우고 다시 버튼을 클릭하면 cors 에러가 출력되어 있을 겁니다.

만약 SOP가 없었을 때 일어나는 보안 공격은, 지금까지도 큰 교훈이 되는 공격 방식입니다.

[27] 파이참에서 라이브 html을 실행할 때 주소이다.

암호의 세계

암호는 정말 다양하게 사용할 수 있습니다. 이제부터 이 암호에 대해 알아보고, 안전한 웹을 구성할 수 있도록 해봅시다.

- 카이사르(시저) 암호

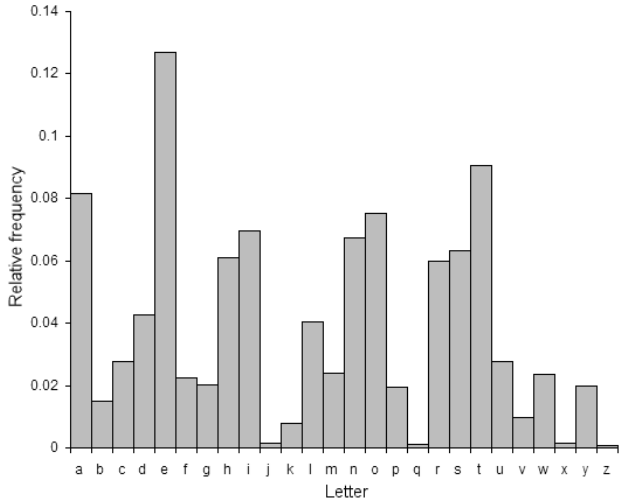
기원전 1세기부터 사용되었던 암호입니다. 간단한 방식이지만, 그때 당시 기준으로 획기적이여서 이후 많은 개량을 거치며 계속 사용되어 왔습니다.

이 암호는 메시지(평문)의 각 알파벳을 특정 횟수만큼 순환이동 시켜서 암호문을 얻는 방식입니다. 반대로 복호화를 할 때는 특정 횟수만큼 앞으로 이동시켜서 평문을 알아낼 수 있습니다. 여기서 이 특정 횟수가 KEY가 되는 겁니다.

당연히 이 암호는 현대에는 안 쓰이지만, 암호의 역사에서 아주 중요합니다. 이 암호가 치환을 설명하기 아주 암호이기 때문이죠. 치환은 각 글자를 다른 어떤 글자로 바꾸는 방식으로 작동합니다. 그냥 이 암호 자체가 암호학에서 치환을 의미하는 것이라고 보셔도 됩니다.
치환은 공통적인 규칙을 가지고 있습니다.

1. 역원이 존재해야 합니다. 정상적으로 복호화를 하기 위해서죠. a b c d가 e e e f라고 바뀌는 경우에는 치환이라고 보기 어렵습니다.
2. 암호학적 치환은 반드시 key로 결정되어야 합니다. 그러면 key의 비밀이 유지하는 한 치환의 비밀도 유지할 수 있습니다.
3. key가 다르면 치환도 다른 게 좋습니다. 만약에 서로 다른 key들이 동일한 치환을 한다면, 이는 서로 다른 key들의 개수보다 치환이 적다는 의미가 됩니다. 카이사르 암호에서 서로 다른 치환은 26개입니다. key는 모든 자연수가 될 수 있습니다만 치환의 종류가 26개밖에 안되므로 실질적으로 26*n + @ 이런 의미입니다.
4. 안전한 치환은 이 의미를 가지고 있습니다. a가 c로 치환될 때, b가 c로 치환되지 않을 것이라는 것만 알 수 있다. 카이사르 암호는, a가 c로 치환이 되었다는 것을 확인하면, b는 d로 치환이 되었다는 것도 알아낼 수 있습니다. 안전한 치환은 이런 게 없어야 한다는 겁니다.

치환 종류가 26가지 밖에 안되므로, 암호 해독을 할 때는 단지 26가지 경우만 생각하면 됩니다. 그럼 어떤 알파벳이 다른 어떤 알파벳으로 치환되었는지(key가 얼마인지)는 암호문만 보고 알아낼 수 있을까요? 알아낼 수 있습니다.
현대 영어에서는 평균적으로 e가 가장 많이 등장합니다. 이를 이용해 가장 많이 등장한 알파벳이 e를 치환한 것이라고 파악하면, 한 번에 해독을 성공할 확률이 높아집니다.(빈도분석법)



- 스키테일

스키테일은 기원전 4세기에 고대 스파르타에서 사용된 암호입니다. 이 암호에서 key는 원통입니다.

종이를 정해진 원통에 감아둡니다. 그리고 이 원통에 글자를 적습니다. 이제 이 종이를 펼치면 암호문이 되는 겁니다. 암호문을 복호화하고 싶으면 다시 원통에 감고, 글자를 읽으면 됩니다. 만약 굵기가 다른 원통이였다면, 제대로 복호화할 수 없을 겁니다.

이 암호문의 재미있는 점은, 글자 중에 바뀐 것은 없습니다. 당연히 글자의 총 길이는 똑같습니다. 다만 이 글자들의 순서가 바뀌었다는 것뿐입니다. 복호화할 때, 같은 원통의 역할은 이 순서를 올바르게 맞추어 주는 역할을 하는 겁니다. 이게 전치법입니다. 앞서 나온

카이사르 암호가 치환을 대표한다면, 이 스키테일은 전치를 대표합니다. 카이사르와 마찬가지로, 현대에 이걸 그대로 쓴다는 건 안되고, 다양한 기술과 융합하여 전치법을 사용합니다.

- 가장 안전한 암호?

절대로 들키지 않는 암호가 존재할까요? 놀랍게도 존재합니다. 일회용 패드입니다. 이 암호는 완전 비밀성을 제공합니다. 아무리 해독을 시도해도, 암호화를 한 쪽에서 실수만 하지 않으면, 절대로 해독을 못합니다. 단지 암호문의 길이와 평문의 길이가 같다는 것만 알 수 있죠.

이 암호와 뒤에서 나올 아주 중요한 것들을 이해하기 위해서는 XOR 연산에 대해 알아두어야 합니다. xor는 배타적 논리합으로, ⊕기호로 나타냅니다. 두 개의 bit에 대한 xor 연산 결과는 다음과 같습니다.
0⊕0 = 0
1⊕1 = 0
1⊕0 = 1
0⊕1 = 1
두 bit 값의 합을 2로 나눈 나머지라고 이해하셔도 됩니다.

일회용 패드는 평문과 평문이랑 같은 길이의 무작위적으로 선정된 bit로 이루어진 key와, xor 연산을 합니다.
CipherText = PlainText ⊕ Key
복호화는 xor 연산의 특징을 이용해 할 수 있습니다.
PlainText ⊕ Key ⊕ Key = PlainText = CipherText ⊕ Key

평문이 011010, Key가 100110일 때, 암호문은 다음과 같다는 겁니다. (두 데이터의 XOR 값)
111100

이름에서 알 수 있는 것처럼, 이 암호는 일회용입니다. 정확히 말하면, key가 일회용이라는 겁니다. 두 암호문이 있을 때, 교환 법칙을 이용하면 쉽게 해독됩니다.
C1 ⊕ C2 = P1 ⊕ K ⊕ P2 ⊕ K = P1 ⊕ P2 ⊕ K ⊕ K = P1 ⊕ P2
이렇게 보안상의 취약점을 줄 수 있기 때문에 일회용 패드는 반드시 한 번만 사용해야 합니다.

수학적으로 안전하다는 것이 보장되지만, 중요한 것은 key와 평문의 길이가 같다는 겁니다. 이는 크기가 큰 데이터에 대해 일회용 패드를 적용시키는 것이 좋지 못한 것임을 알 수

있습니다. 그래서 현대의 암호는 현실적으로 사용이 가능하면서, 해독을 어렵게 하는 방식들 위주 입니다.
참고로 현대의 암호는 대부분 알고리즘이 전부 공개되어 있습니다. 그래서 제가 이 챕터를 쓸 수 있었겠죠. 전적으로 암호는 key에 의존해야 합니다. 현대 암호는 다른 모든 것을 알아도 key를 모른다면 암호 해독을 할 수 없게 한다는 목표를 가지고 암호를 개발합니다.

- DES

1979년부터 표준으로 사용었다가 21세기에 접어들며 퇴출된 블록 암호입니다. 먼저 블록 암호의 전체적인 내용부터 알아보고 들어가도록 합시다.
기본적으로 블록 암호도 대칭키 암호이기 때문에 하나의 암호화, 복호화 알고리즘을 가지고 있습니다. 암호화 알고리즘(E)은 Key와 평문 블록 P를 받아서 암호문 블록 C를 만들어 냅니다. 간단하게 C = E(Key, P)라고 표기하겠습니다. 복호화 알고리즘(D)은 암호화 알고리즘의 역에 해당합니다. 그래서 보통 두 알고리즘은 비슷한 경우가 많습니다.
간단하게 P = D(Key, C)로 표기하겠습니다.
여기서 핵심은 블록입니다. 평문이 들어오면 일정한 크기의 블록으로 나눈 후, 이에 대해 암호화, 복호화 알고리즘을 적용하는 겁니다. 블록의 크기는 짧으면 보안이 약하고, 너무 길면 메모리에 부담이 갑니다. 유명한 블록 암호에서 블록 크기는 보통 64 ~ 128bit 정도 됩니다.
만약 평문이 딱 나누어 떨어지지 않거나, 각 블록 간에 특별한 연결성 같이 세세한 부분들은, 운용모드라는 것이 달라지게 됩니다.
각 블록은 일련의 라운드를 계산하는 것으로 구성됩니다. 쉽게 표기하자면, 각 라운드 알고리즘이 R이고, 이 라운드마다 사용되는 key가 다를 때, 다음과 같이 나타낼 수 있습니다.

C = R3(K3, R2(K2, R1(K1, P)))

평문과 첫 번째 라운드 키를 이용해 첫 번째 라운드 값을 구하고,
이 값과 두 번째 라운드 키를 이용해 두 번째 라운드 값을 구하고,
이 값과 세 번째 라운드 키를 이용해 최종 암호 블록의 값을 구합니다.

이 것까지 블록 암호의 개념이였고, 현대 암호에서 미리 좀 더 알아야 하는 내용이 있습니다.
안전한 암호는 혼돈과 확산이 커야 합니다.
혼돈과 확산을 설명하는 방법이 다양하니, 제 설명이 이해가 안되시면, 다른 비유 등을 들어 설명하는 것을 찾아보셔도 도움이 됩니다.
먼저 혼돈이란, 평문과 암호문의 상관관계를 없애는 겁니다. 즉 입력된 평문과 key가 복잡한 변환 과정을 거치게 된다는 것이죠. 확산은 hello와 hello,의 암호문이 서로 완전히 다르게 생성될수록 크다고 합니다. 즉, 평문과 암호문 사이 통계적 분석을 어렵게 하는 것이죠.

흔히 P-box 이것 자체로는 암호가 되지는 않지만, 다른 기능들과 합쳐져 안전한 암호를 만들어 나가는 기능(전치)을 합니다.

3	1	4	2

이런 형태의 s-box가 있다면, abcd가 s-box에 적용될 때, 다음 결과가 나올 겁니다. cadb. 원래대로 되돌려 놓을 때는 이 표를 사용하면 됩니다.[28]

2	4	1	3

이것 말고도 대입 상자, S-box라고 부르는 것이 있습니다. 이건 치환의 역할을 합니다. 물론 이것도 단순히 수많은 기능들과 결합하는 것들 중에 하나뿐이지만, 자주 강조됩니다. 아무거나 s-box로 사용하는 건 아니고, 최대한 규칙이 없어보여야 제대로 사용할 수 있습니다.

입력을 둘로 나눔	1	2	3	4
a	b	a	e	d
b	c	g	f	h

1a가 입력으로 들어왔으면 b, 4b가 입력으로 들어왔으면 h 등 치환 역할을 해줍니다. 이것 외에도 특정 수학 공식을 적용하여 치환을 하는 등 s-box는 다양합니다.

이제 본격적으로 DES에 대해 알아보도록 하겠습니다.
FIPS 46-3, Data Encryption Standard (DES) (withdrawn May 19, 2005) des는 파이스텔 구조라고 부르는 것을 기반으로 합니다. 파이스텔 구조의 핵심은 블록을 둘로 나누어서, 작동시킨다는 겁니다. 바로 des의 동작 과정을 살펴보도록 하겠습니다.

먼저 des는 56bit의 key를 사용하며, 64bit로 블록을 구성하고, 16번의 라운드 과정을 거칩니다.
각 블록에서 일어나는 암호화 과정은 다음과 같습니다.

1. 사용자에게 입력받은 key를 가지고 각 라운드 key를 생성합니다.
    A. 56bit의 key를 각각 28bit씩 둘로 나눕니다.
    B. 이 두 절반짜리 블록은 각각 미리 지정되어 있는 P-box에 의해 전치가 됩니다.
    C. 그리고 각각 왼쪽 순환 이동이 됩니다. 10011 이런식이면 00111 이렇게 되는 겁니다.
    D. 1, 2, 9, 16번째 라운드에서 사용할 key는 1bit씩 순환 이동을 합니다. 나머지는 전부 2bit씩 순환 이동을 합니다.

    이 구조에 맞춰서 비유를 해보면, 블록이 나뉘고 전치과정을 거쳤을 때, 어느 한쪽의 값이 110101100이였을 때,

[28] 두 표 사이의 관계를 잘 생각해봅시다.

1라운드 결과 101011001 (하나 왼쪽으로 이동)

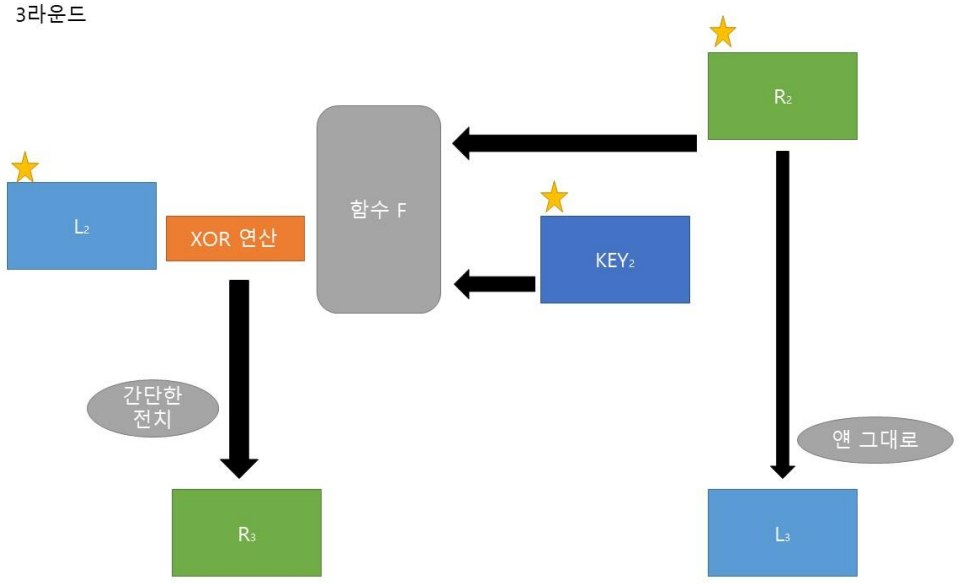
2라운드 결과 010110011 (1라운드 결과에서 하나 왼쪽으로 이동)

3라운드 결과 011001101 (2라운드 결과에서 둘 왼쪽으로 이동)

E. 이 과정을 거치고 나온 두 절반짜리 블록이 다시 합쳐집니다. (16라운드 key는 순환이동 16번 다 하고 나오는 것이고, 1라운드 key는 그냥 첫 번째 순환 이동만 하고 나오는 겁니다.) 그리고 다시 다른 p-box에 의해 전치가 됩니다. 이때 전치는 축소 전치로, 56칸이 아닌 48칸 입니다. 중간에 순서가 섞이면서 중간에 빠지는 bit들이 있는 것이죠.

이렇게 16개의 48bit 라운드 key들이 완성됩니다.

2. 64bit의 평문 블록은 먼저 한 번 p-box에 의해 전치가 됩니다. 그리고는 L과 R이라 불리는 두 절반짜리 블록으로 나뉘게 됩니다.
3. 첫 번째 R은 그대로 다음 라운드(두 번째) L이 됩니다.
4. 첫 번째 L은 복잡한 과정을 거쳐 다음 라운드(두 번째) R이 됩니다.
5. 이 과정을 16번 반복하면 됩니다. L0과 R0에서 시작하였다면, 결과로는 L16과 R16이 나올 겁니다.[29]
6. 그리고 이 둘을 합치고 전치를 합니다. 전치를 할 때 사용되는 p-box는 맨처음 평문 전체 블록에서 사용하였던 것의 역 p-box입니다. p-box를 설명할 때 나왔던 두 표 사이의 관계와 똑같은 겁니다.
7. 하나의 블록에 대한 암호화가 모두 진행되었습니다.



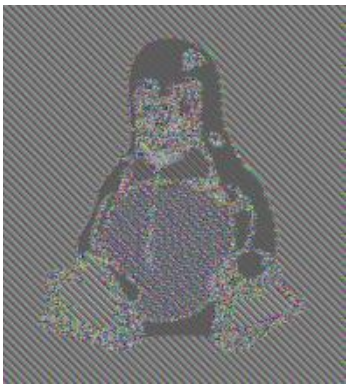
4번 과정에서 나온 복잡한 과정(F함수)에 대해 자세히 알아보도록 하겠습니다.

[29] 저도 처음에는 헷갈렸는데, 만약 R1과 L1에서 시작하여서 R16과 L16이 된다면, 라운드(과정)가 15밖에 안됩니다. 그래서 R0과 L0부터 시작하는 겁니다.

1. R과 라운드 key가 F함수의 인수로 들어가게 됩니다.
2. R은 먼저 확장 전치를 합니다. 앞서 라운드 key에서 나온 축소 전치의 반대입니다. 확장 전치 p-box를 이용해 32bit 데이터를 48bit로 늘립니다.
3. 확장 전치된 R과 라운드 key가 서로 XOR 연산을 합니다.
4. XOR 연산을 한 결과는 s-box에 들어갑니다. s-box에서는 48bit가 6bit씩으로 나뉘게 됩니다.
5. s-box에서는 6bit를 4대 2로 나누어서 표에 적용시키고, 치환하게 될 값을 얻게 됩니다. 이때 치환한 값은 4bit입니다.
6. 6bit씩으로 된 8개의 데이터가 모두, 치환할 4bit의 값을 얻게 되면, 다시 합칩니다.
7. 이렇게 32bit의 데이터를 얻게되고, 이 데이터와 L이 XOR 연산을 합니다
8. xor 연산 값이 다시 p-box를 통과하게 되어 다음 라운드에서 사용될 R이 됩니다.

아주 조금 생략한 부분이 있긴 하지만, DES는 전체적으로 이렇게 작동한다고 이해하시면 됩니다. 복호화 과정은 따로 설명하진 않았지만, 그냥 왔던 길을 다시 되돌아가면 됩니다.
이제 이 암호화된 평문 블록을 어떻게 하냐에 따라 운용모드가 나뉩니다.

ECB:
그냥 암호화된 블록끼리 서로 합칩니다. 그렇게 전체 암호문을 만드는 것이죠. 매우 간단한 대신 패턴이 유출될 수 있다는 것이 단점입니다. 빈도분석법하고 비슷한 것이죠.


리눅스 캐릭터 그림을 ECB로 암호화하고, 다시 그 그림을 보면 ECB 모드의 한계를 직관적으로 익히실 수 있습니다. 실제 웹 서비스에서 암호를 사용하신다면, ECB는 모드는 사용하시면 안됩니다.

CBC:
블록을 암호화하기 전에 이전 암호 블록과 XOR 연산을 합니다. 그 결과로 나온 것으로 암호화를 진행하는 것이죠.
첫 번째 블록은 미리 지정되어 있는 데이터와 XOR 연산을 한 후 암호화를 하고,
두 번째 블록은 첫 번째 블록이 암호화된 것과 XOR을 한 후 암호화를 하고,
이런 식으로 진행되는 것입니다.
안전한 대신 중간에 어떠한 결함으로 bit가 바뀔 경우, 전체에 악영향을 줄 수 있습니다.

ECB는 문제가 발생해도 한 블록에만 영향을 끼치기 때문에 상대적으로 안전합니다.
그래도 실제 서비스에서 CBC모드가 가장 안전하면서 효율적이라서 많이 사용합니다.

이외에도 블록 암호의 개성을 없애버리는 CTR 모드, 인기가 너무 없는 OFB 모드 등이 있습니다.

DES는 알고리즘에 조금씩 약한 취약점이 있다는 것이 밝혀지고 있습니다. 그러나 DES가 사용 비권장이 되는 주원인은 무차별 대입 때문입니다. 가능한 key의 경우는 2^{56}뿐입니다.
현재 개인용 PC의 평범한 사양으로도, 금방 해독할 수 있는 수치입니다.
DES를 3번 연속으로 돌리는 3DES도 있지만, 뒤에 나올 AES에게 모든 부분에서 밀려서 결국 DES는 역사 속으로 들어가게 되었습니다.[30]

다양한 암호에서 수학적인 방법으로 아주 현명하게 해독을 하는 경우도 있지만, 쉬워보이는 무차별 대입도 아주 큰 결과를 가져올 수 있습니다.

- AES

현재 모든 소프트웨어에서 가장 많이 쓰인다고 할 수 있는 블록 암호입니다. 아마 웹 서비스에서 대칭키 암호를 도입해야 한다면, 최우선적으로 고려될 겁니다.[31] 2000년부터 지금까지 NIST(미국국립표준기술연구소)에서 지정한 표준입니다.

FIPS 197, Advanced Encryption Standard (AES)

1997년부터 암호 알고리즘을 공모 받은 결과, 벨기에에서 출품된 레인달이라고 불리는 알고리즘으로 된 암호입니다.

key는 128bit, 192bit, 256bit 중 하나를 사용할 수 있고, 이에 따라 라운드 수도 10, 12, 14로 바뀝니다.
그리고 파이스텔 구조가 아닌 SPN 구조를 사용합니다. 한 블록은 16Byte (128bit)를 담고 있습니다. bit로 안하고 byte로 하는 이유는 블록들을 행렬로 나타내기 때문입니다.
4*4행렬로 나타내고, 보통 16진수 문자 2개를 한 칸에 넣는 것으로 표현합니다. 하나의 16진수는 4개의 bit를 완전히 표현해줄 수 있는 문자이거든요.

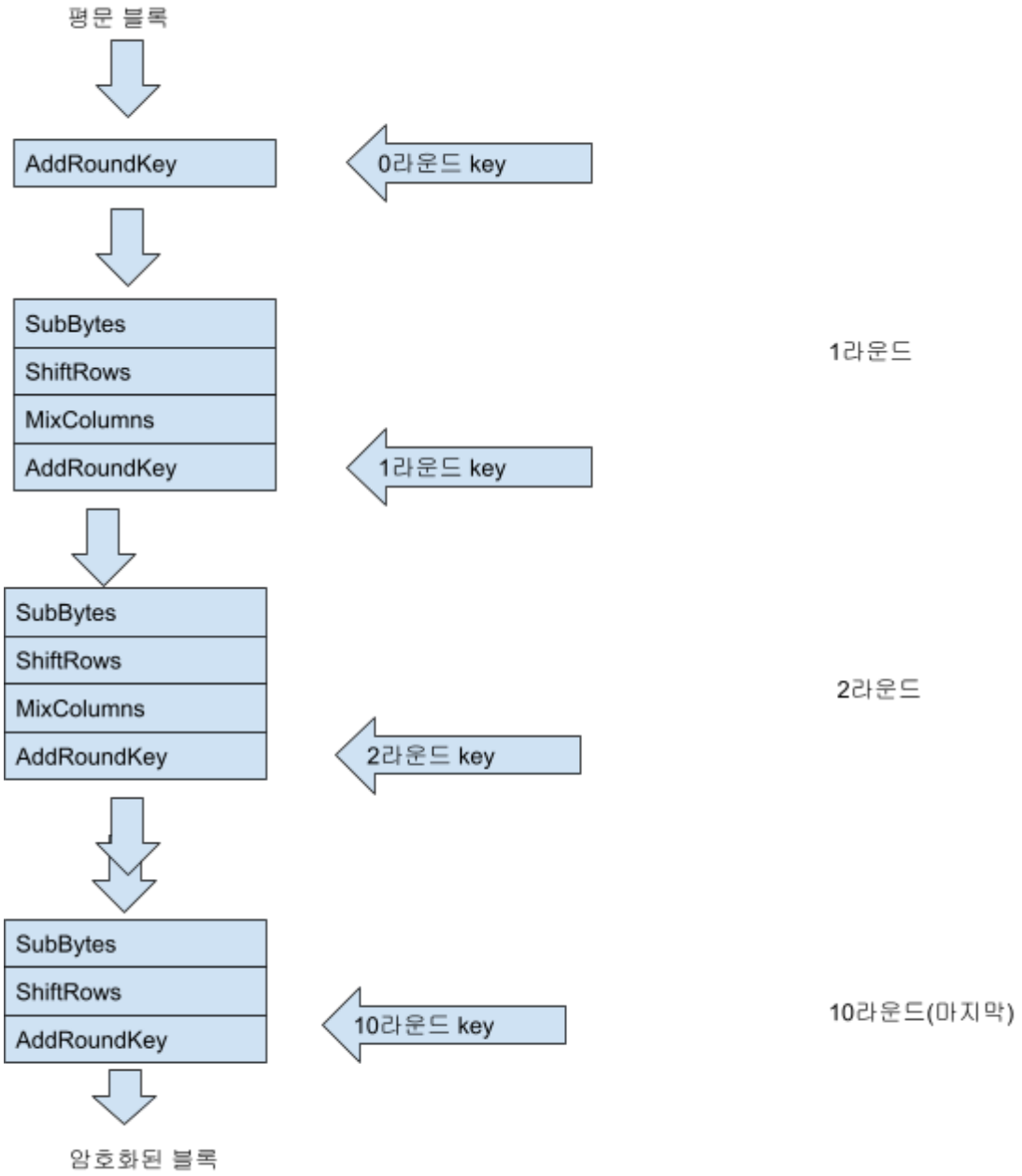
1 9	a a	2b	4a
2 5	6 7	4 a	5 c

[30] 혹시라도 아직도 실제 웹 서비스에서 암호화에 DES를 사용하고 있다면, 개발자를 비롯한 웹 사이트 관련 인물들을 경계하시는 게 좋을 듯 합니다.
[31] 저도 마찬가지 입니다.

f 1	e 3	3 a	2 e
d a	c 1	1 3	7 9

이런 식으로 한 블록을 표현합니다.

전체적인 알고리즘의 과정은 다음과 같습니다. 128bit key 기준입니다.



AddRoundKey:

그냥 라운드 key(128bit)와 블록(128bit)끼리 XOR를 하여 결과를 만들어 냅니다. 보안성을 높이기 위해 라운드 시작 전에 한 번 추가로 합니다.(0라운드)

SubBytes:

앞서 나온 블록 행렬에서 한 칸씩 s-box에 적용시켜서 치환을 하는 작업입니다. 가로와 세로 각각 16개씩인 이 s-box 표에, 한 칸에 들어가는 두 16진수 문자를 이용하여 치환할 값을 구합니다. 그리고 원래 칸에 그 값을 대신 넣어서 치환을 하게 됩니다. 참고로 사실은 어떤 알고리즘에 따라 이 치환할 값이 정해지는 겁니다. 그냥 표(s-box)로 만들어서 사용하는 것이 편해서 흔히들 s-box라고 이야기 합니다.

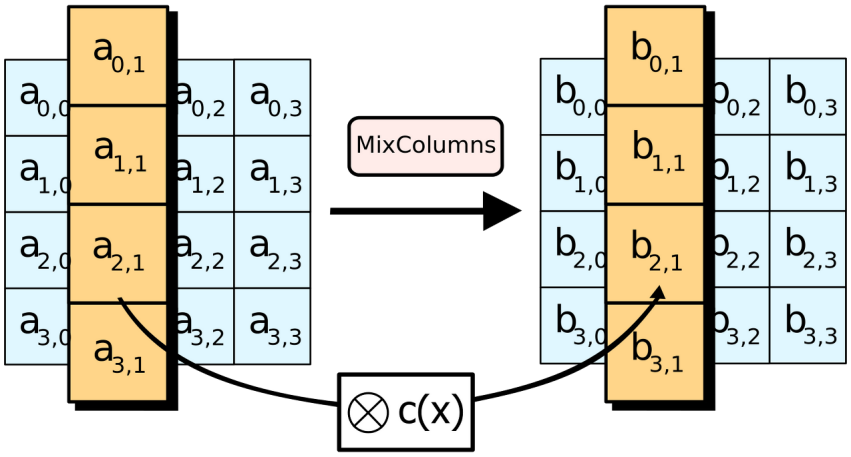
ShiftRows:

블록 행렬에서 가로줄(행)에 있는 데이터들을 순환이동시킵니다. 첫 번째 가로줄은 가만히 있고, 두 번째는 1번, 세 번째는 2번, 네 번째는 3번 순환이동 시킵니다.
수식적으로 말하자면, 0부터 3까지의 행에 대해 i행을 왼쪽으로 i만큼 순환이동 시킨다라고 표현할 수 있겠습니다. 위에서 나온 예시 행렬을 이용하면 다음과 같습니다.

1 9	a a	2b	4a
6 7	4 a	5 c	2 5
3 a	2 e	f 1	e 3
7 9	d a	c 1	1 3

MixColumns:

세로줄이 하나로 묶여서 어떤 연산이 일어나게 됩니다. 미리 준비되어 있는 4*4 행렬과 세로줄이 하나씩 특수한 연산을 하여, 세로줄 하나의 값을 얻어내게 됩니다. 이 값을 원래 세로줄에 대체하는 것이죠. 마지막 라운드에서는 이 것이 생략됩니다.



복호화를 하려면, des처럼 key를 가지고 왔던 길을 다시 되돌아가면 됩니다.

aes에서 128bit의 key로부터 각 라운드 key를 만드는 과정은 다음과 같습니다.
먼저 key를 16byte의 4*4 행렬 형태로 나타냅니다.

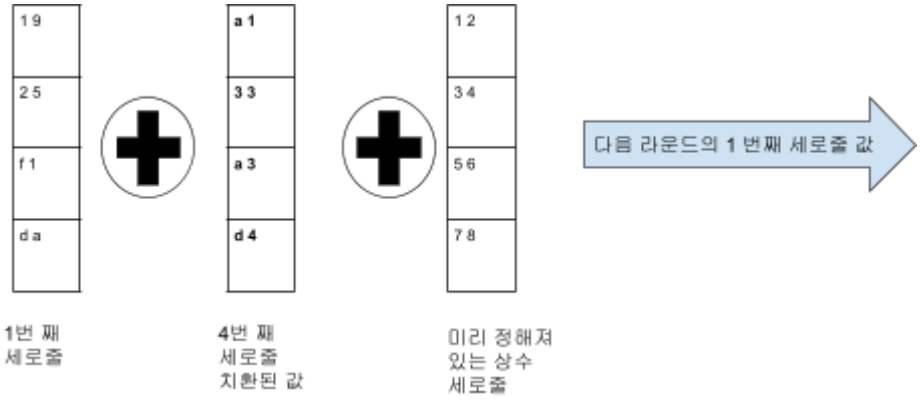
어떤 key가 다음과 같다고 해보겠습니다.

1 9	a a	2b	4 a
2 5	6 7	4 a	5 c
f 1	e 3	3 a	2 e
d a	c 1	1 3	7 9

먼저 4번 째 마지막 세로줄(열)을 한 칸씩 위로 순환이동 합니다. 다음 표를 예시로 들어보면

5 c
2 e
7 9
4 a

이제 이걸 s-box에 의해 같은 크기의 세로줄로 치환을 합니다.
이 치환을 한 값과 1번 째 세로줄과 xor를 합니다.
그리고 또 미리 정해둔 상수 세로줄과 한 번 더 xor를 합니다.



이렇게 라운드 key의 1번 째 세로줄 값을 구했습니다.

2번 째 세로줄은 방금 만든 라운드 key 1번 째 세로줄과, 전체 key의 2번 째 세로줄이 xor한 값을 넣습니다.
3번 째 세로줄도 방금 만든 라운드 key 2번 째 세로줄과, 전체 key의 3번 째 세로줄이 xor한 값을 넣습니다.
4번 째 세로줄도 방금 만든 라운드 key 3번 째 세로줄과, 전체 key의 4번 째 세로줄이 xor한 값을 넣습니다.
이제 처음 라운드 key를 다 만들었습니다.
다음 라운드의 경우, 이 과정과 거의 동일합니다. 다음 라운드 key부터는 원래 key로 부터 생성하는 것이 아닌, 바로 앞 라운드 key를 이용하여 동일한 방법으로 만듭니다.

다른 길이의 key를 사용하더라도 결과적으로 라운드 key는 128bit로 모두 동일합니다.
그리고 라운드 시작 전에 한 번 라운드 key와 블록이 xor를 하기 때문에 전체 라운드+1만큼의 라운드 key가 필요하다는 것을 잊지 말아야 합니다.
aes의 알고리즘에 들어있는 모든 것은 다 이유가 있습니다. 현재까지 유의미한 해독 공격이 발견되지 않았고, 앞으로도 그럴 겁니다. 웹 서비스에서 대칭키 암호를 적용해야 한다면, aes가 가장 최적일 겁니다. 그러나 ECB 같은 운용모드를 장기간 운용하는 것은 aes의 뛰어난 보안 능력이 무력화될 수 있으니, 운용모드도 신경써야 합니다.
아주 뛰어난 암호이지만, 일회용 패드와 달리 수학적으로 안전하다는 것이 완전히 증명되지는 않았습니다. 다면 현재 인류가 알고 있는 공격들에서 모두 안전하다는 것만 알죠. 극단적으로 수백 년간 안전할 수도 있고, 바로 내일 이런 뉴스가 나올 수도 있죠.
*AES, XXX*가 개발한 최신 해독 알고리즘에 의해 완전히 무력화. 보안 업계 혼란
그래도 암호는 공격을 받을수록, 더욱 강하다는 것이 조금씩 증명됩니다. 웹 서비스를 개발, 운영하시면서 어느날 보안 관련 소식지에서 사용하는 것이 위험하다고 하면, 그 때 새 암호로 바꾸면 되는 겁니다.[32]

```
from base64 import b64encode, b64decode
from binascii import unhexlify
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

# 16진수 넣어야 됩니다
iv = "7bde5a0f3f39fd658efc45de143cbc95"
password = "11111111111111111111111111111111"

msg = "this is a message"

print(f"IV라고 부르는, CBC모드에서 첫 번째 블록과 xor를 해주는 상수입니다. {iv}")
print(f"KEY {password}")
print(f"평문 {msg}\n")

iv = unhexlify(iv)
password = unhexlify(password)
msg = pad(msg.encode(), AES.block_size)
cipher = AES.new(password, AES.MODE_CBC, iv)
cipher_text = cipher.encrypt(msg)
out = b64encode(cipher_text).decode('utf-8')
print(f"암호문 {out}\n")
```

[32] 역사적으로 보았을 때 실제 문제를 일으키기 전에 보통 다른 암호를 쓰라고 안내가 나왔습니다. 그때 바꾸시면 충분합니다. 몇 년 째 바꾸어야 한다고 하는데도 안 바꾸다가 큰일 나는 경우도 있습니다.

```
decipher = AES.new(password, AES.MODE_CBC, iv)
plaintext = unpad(decipher.decrypt(b64decode(out)),
AES.block_size).decode('utf-8')
print(f'암호문을 복호화한 것 {plaintext}')
```

pip install pycryptodome

간단한 파이썬 라이브러리를 사용하여 aes cbc 모드를 이용해 텍스트를 암호화 한 것입니다.

라이브러리가 많은 파이썬 외에도 다양한 언어에서 aes 라이브러리가 많이 있으니, 잘 활용하시면 될 것 같습니다.

- 스트림 암호

대칭키 암호는 블록 암호 아니면 스트림 암호입니다. 스트림 암호는 블록으로 나누지 않고, 한 번에 xor 등의 연산을 통해 암호화를 하는 대칭키 암호입니다. 작동 방식을 보면 일회용 패드와 매우 비슷해 보입니다.

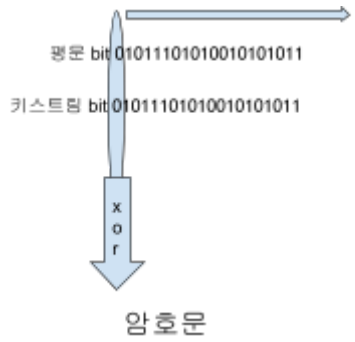
웹 개발에서는 스트림 암호는 보기 힘든 편입니다. 같은 대칭키 암호인 블록 암호에 비해 인기가 떨어지기 때문입니다. 인기가 없는 이유는 역사적인 부분에 있습니다.
역사적으로 스트림 암호들은 블록 암호보다 보안성이 매우 떨어졌습니다. 단순히 개인이 제작한 암호뿐만 아니라 표준으로 사용되던 암호들도 허무하게 해독당한 사례가 있었습니다. 또한 현재 AES가 완전한 표준으로 자리 잡았기 때문에, 작동 방식이 다른 스트림 암호가
상대적으로 밀리는 것 같습니다.

그러나 다양한 시스템에 내장되어 우리 생활에 많이 쓰입니다. 특히 무선 통신에 많이 사용되는 데, 이는 스트림 암호가 블록 암호보다 구조적으로 빠를 수 밖에 없다는 것 때문입니다. 무선 통신에서는 보안성과 속도를 모두 맞추어야 해서, 스트림 암호가 잘 쓰입니다.
아주 중요한 데이터를 다루는 것이 아닐 때, 일반 상용 소프트웨어(웹 서비스)에 최적화된 스트림 암호[33]가 많이 사용되고 있는 추세입니다.

공통적인 스트림 암호들의 구조는, key 스트림과 평문이 xor 연산을 한다는 겁니다. 그 결과로 암호문이 나오게 됩니다. 여기서 만약 key 스트림 자체가 key라면 그것은 일회용 패드가 될 겁니다. 그러나 스트림 암호는 key를 입력을 받고, 특수한 알고리즘에 따라 key 스트림을 생성합니다. 많은 스트림 암호들이 key 스트림 생성 알고리즘에서 취약점이 나타나게 되었습니다.

[33] 당연히 별도의 취약점이 발견되지 않은 견고한 스트림 암호들 입니다.

앞서 나왔던 것처럼 생각해보면, key 스트림은 다시 사용해서는 안됩니다. 그런데 같은 key를 다시 사용하고 싶다면요? 그래서 key 스트림 생성 알고리즘에서는 nonce라고 부르는 것을 알고리즘의 인수로 사용합니다.



다양하게 스트림 암호를 분류하긴 하지만, 대표적으로 동기식, 비동기식으로 나누는 것이 있습니다.

동기식 스트림 암호

key 스트림이 평문과 아무 관련이 없는, 즉 사용자의 입력으로만 key 스트림울 생성하는 방식입니다. 암호문에서 새로운 bit가 추가되거나, 기존의 bit가 사라지게 될 경우(추가, 삭제) 완전히 에러가 나옵니다.

자기(비) 동기식 스트림 암호

key 스트림이 평문(복호할 때는 암호문)에 종속적입니다. 즉, 평문(암호문)이 key 스트림을 결정하는 데 영향을 줍니다. 수정뿐만 아니라 추가, 삭제 상태에서도 완전한 에러가 생기지 않고, 일부분만 문제가 생기는 구조입니다. key 스트림과 평문이 함수 관계를 가지고 있으므로 완전한 에러를 피할 수 있습니다.(자기 동기성)

아마 웹 개발에서 스트림 암호를 사용하시겠다면, salsa20이 크게 고려될 겁니다.
[Salsa20 - eSTREAM Phase 3]
현재까지 알려진 구조적으로 유효한 공격이 없는 동기식 스트림 암호입니다.
pip install salsa20

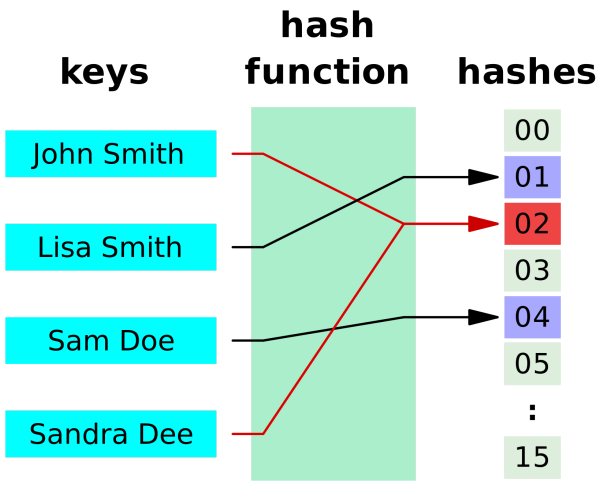
```
from salsa20 import XSalsa20_xor
from os import urandom

nonce = urandom(24)
KEY = b'*secret**secret**secret**secret*'
```

```
# 암호화
ciphertext = XSalsa20_xor(b"IT'S A YELLOW SUBMARINE", nonce, KEY)
# 한 번 더 xor 연산을 하여 복호화
print(XSalsa20_xor(ciphertext, nonce, KEY).decode())
```

간단한 salsa20 파이썬 라이브러리입니다. 이 자체 라이브러리 외에도 pycryptodome에서도 여러 스트림 암호들을 지원합니다. 아주 중요한 데이터가 아닌데, 암호화가 필요할 듯 하면 스트림 암호는 괜찮은 선택일 겁니다.

- HASH

해시는 앞에서 파일 업로드 취약점에서 나온 대책 방안 중 하나입니다. db에서 사용자가 입력한 파일 이름하고, 서버에 실제 저장된 파일 이름(해시 값)하고, 연결해주는 과정이 해시 테이블인 것이죠.



이건 간단한 해시 테이블이지만, 지금 나오는 쓰이는 해시는 아주 엄격합니다.
항상 동일한 길이를 보장합니다.

어떤 자연수 x를 10으로 나눈 나머지가 항상 동일한 길이를 보장하는 것도 하나의 해시 함수입니다.

어떠한 수를 10으로 나누더라도, 나머지는

0 (x = 0, 10, 20 등)

1 (x = 1, 11, 21 등)

2 (x = 2, 12, 22 등)

3 (x = 3, 13, 23 등)

4 (x = 4, 14, 24 등)

5 (x = 5, 15, 25 등)

6 (x = 6, 16, 26 등)

7 (x = 7, 17, 27 등)

8 (x = 8, 18, 28 등)

9 (x = 9, 19, 29 등)

중 하나가 반드시 나옵니다. 즉 항상 동일한 길이를 보장합니다.

앞에 나온 다른 암호들은 평문을 암호화해서 저장하거나 전송해서 그 자료를 공격자가 알아내지 못하도록 하는 데 주력하였습니다. 그러나 해시 함수는 자료가 수정되었을 때, 그것을 알 수 있게 만들어줍니다.

그래서 안전한 보안 해시 함수는 어떤 두 자료에 대해 다른 출력 값을 내놓아야 합니다. 이외에도 같은 자료에 대해서는 항상 같은 값을 출력으로 내놓아야 합니다. 그리고 해시 값으로부터 수학적인 방법을 통해 원래 평문을 알아낼 수 없게 해야 합니다.

위 이미지에서 서로 다른 입력인데 해시 값이 같은 것이 있습니다. 이럴 경우 충돌이 발생하였다고 합니다. 충돌은 피할 수록 좋지만, 완전히 피하는 것은 불가능합니다.
해시 입력으로는 다양한 데이터들이 들어갈 수 있습니다. 그러나 출력 값은 수가 제한되어 있습니다.(길이가 정해져 있어서) 수학적으로 반드시 같은 출력을 가지는 데이터가 있을 수 밖에 없습니다.

역상 저항성
어떤 해시 값 h가 있습니다. 이 h를 출력으로 내놓는 데이터 m을 h의 역상이라고 부릅니다.
역상 저항성은 어떤 해시 값이 주어졌을 때, 공격자가 그 해시 값의 역상을 찾아낼 수 없다는 속성입니다. 안전한 해시 함수는 역상 저항성이 커야 합니다.
역상 저항성이 클수록 일방향 함수의 특징이 커집니다.(반대 방향으로 변환 불가능)

- 제1역상 저항성: 위에서 설명한 겁니다. 해시 값이 주어졌을 때, 현실적으로 이 역상 값들을 찾아낼 수 없게 해야 한다는 속성입니다.
- 제2역상 저항성: 어떤 데이터 m이 주어졌을 때, 동일한 해시 값을 출력하는 데이터 n을 찾을 수 없게 해야 한다는 속성입니다.

제1역상 저항성이 공격당하면(제1역상을 구할 수 있으면)
제2역상 저항성도 공격 당합니다.(제2역상도 구할 수 있습니다)
반대로 말하면,[34] 제2역상 저항성이 보장된다면, 제1역상 저항성도 보장됩니다.

[34] 명제의 대우(contraposition)입니다.

충돌 저항성
충돌을 찾아내는 것이 원래의 메시지를 찾아내는 것만큼이나 어려워야 합니다. 위의 역상 저항성하고 비슷한 내용인데, 만약 어떤 해시 함수의 제2역상을 찾을 수 있다면 충돌도 찾아낼 수 있는 겁니다.

안전한 암호학적 해시 함수(알고리즘)는 유용하게 사용할 수 있습니다. 어떤 데이터를 보낼 때, 해시 값과 같이 보냄으로써 전송 중 데이터가 손상되었는지 확인할 수 있습니다.(1bit만 달라도, 크게 다른 해시 값이 나오니까요.)
이외에도 굳이 복호화할 필요가 없거나, 복호화되면 안되는 데이터도 해시 함수를 이용합니다. 주로 비밀번호 등이 DB에 저장될 때, 해시 값으로 저장됩니다. 아주 중요한 데이터를 비교할 때는, 잘못하면 외부로 유출될 수 있는 방법인 데이터 원본끼리의 비교보다는 해시 값을 이용한 비교를 추천합니다.

암호학적 해시 함수가 엄청난 수학적 위협을 물리쳐도, 더 강력한 보안 공격에 당할 수 있습니다.
사람들이 자주 사용하는 데이터(비밀번호 등)의 해시 값을 미리 구해놓고, 나중에 그 해시 값을 검색해서 원래 값을 알아내는 레인보우 테이블 공격입니다.

CrackStation

무차별 대입에서 나왔던 사전 공격과 비슷한 겁니다.
이 공격을 예방하는 방법은 salt를 사용하면 됩니다. salt라고 불리는 임의의 값을 원래 데이터의 앞이나 뒤에 붙이고, 해시 함수로 돌리는 겁니다. salt는 외부로 유출될 경우, salt를 첨가한 레인보우 테이블에 당할 수 있으니 개발자 외에는 salt 값을 알아내서는 안됩니다.

현재 다양한 암호학적 해시 알고리즘이 나와있습니다.
md5나 sha1은 해시 값의 길이가 너무 짧아 공격에 취약해집니다.
최소한 sha2로 사용합니다. 더 안전하게 사용하려면 sha3나 blake2 정도는 되어야 합니다.

```
import hashlib


word = b"hello"
# 솔트 첨가
word = b'this_is_salt' + word
# 사용하지 말 것
print(hashlib.sha1(word).hexdigest())
```

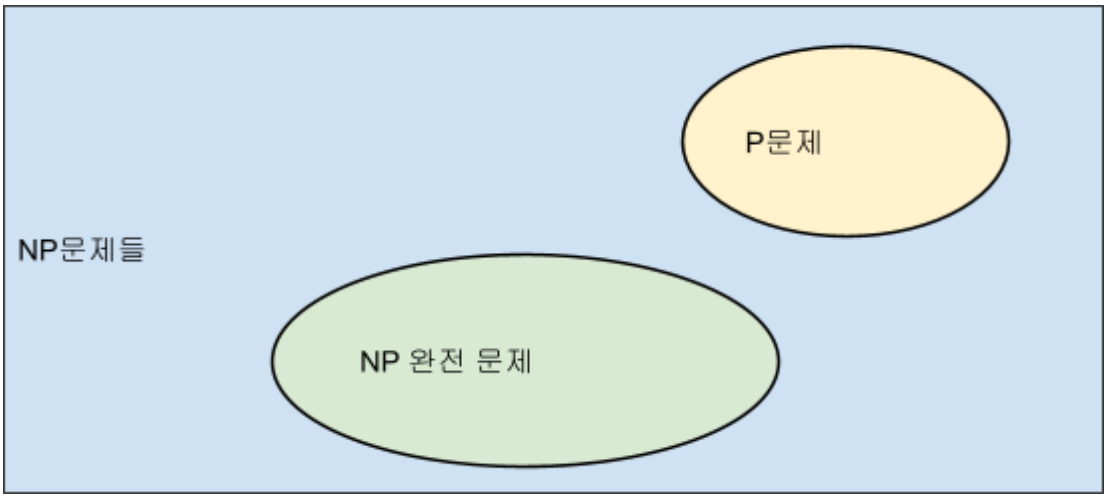
```
print(hashlib.md5(word).hexdigest())
# 기본
print(hashlib.sha256(word).hexdigest())
# 안전
print(hashlib.blake2b(word).hexdigest())
print(hashlib.sha3_512(word).hexdigest())
```

파이썬의 경우 내장 모듈로 암호학적 해시 함수를 제공합니다.

- 비대칭키 암호

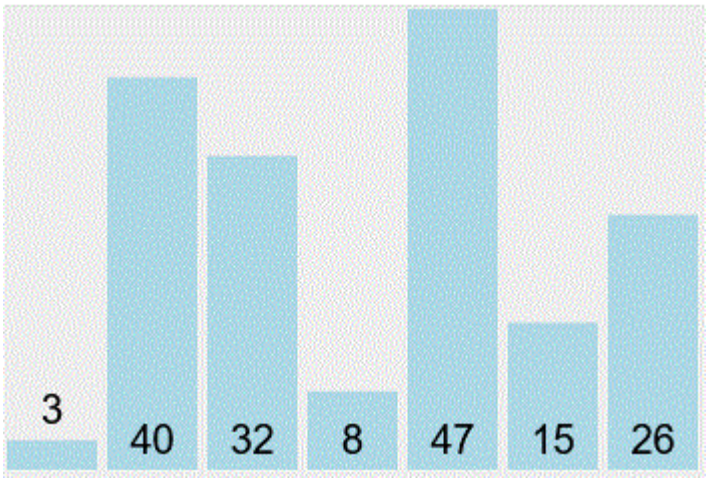
할 내용도 많고 아주 어렵습니다.

먼저 np-p 문제에 대한 이해가 있어야 합니다.

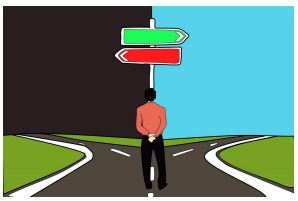


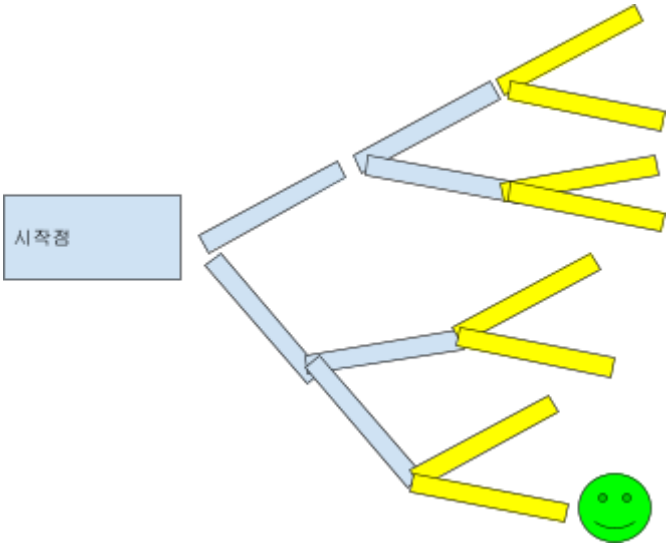
np-p 문제는 증명되지 않았지만, 주요 추측에 의하면 위와 같은 포함 관계를 가지고 있을 것이랍니다.

np, p에 관한 문제를 보면 이해가 쉽지 않습니다. 간단한 예시로 시작하도록 하겠습니다.



이건 버블 정렬 알고리즘을 입체적으로 표현한 겁니다. 이 알고리즘에서 정렬해야 할 원소가 n일 때, 최악의 경우에서 수행해야 할 과정(여기에서는 두 값의 위치 교환)이 n*(n-1)/2이 될 겁니다.
이때, 이 n*(n-1)/2은 n에 대한 다항식입니다[35]. 버블 정렬 알고리즘은 n*(n-1)/2내에 해결이 가능하니, 다항시간 내에 풀 수 있는 p 문제가 됩니다.


어떤 갈림길을 지나야 한다고 해보죠.
이 갈림길이 연속으로 연결된 횟수를 n이라 합니다.
예를 들어 n이 3일 때는 다음과 같습니다.

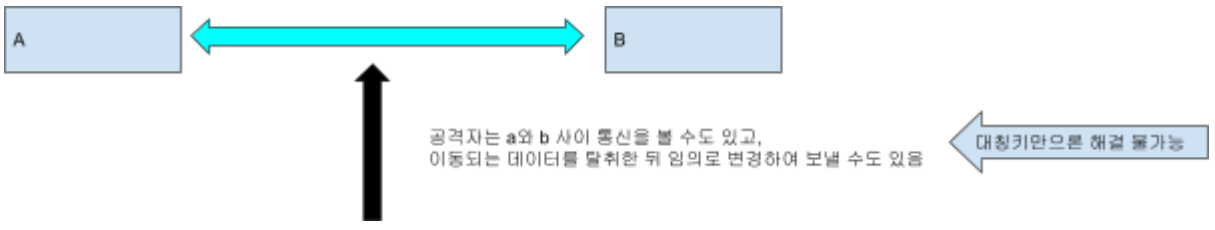


총 도착 지점이 8개입니다. 만약 여기 중에서 정답인 갈림길을 찾는 문제라고 해보죠.
정답을 찾기 위해서는 갈림길로 들어가야 합니다. 노란 색 갈림길로 들어가봐야 하는 최대 횟수는 2^n 입니다. 더 직관적인 예시를 들기 위해 정확히 시간으로 계산한다고 하겠습니다. 위에서 한 스틱을 이동할 때 1분이 소요됩니다. 그리고 마지막에 도착하고 나면 무조건 다시

[35] O 표기법에서는 상수를 모두 생략하니 $O(n^2)$로 나타낼 수 있습니다.

시작점으로 돌아옵니다. 그러면 왕복 2*n분이 소요될 겁니다. 정답을 찾기 위한 이 과정에서 최대로 걸리는 시간이 $2^n * 2 * n$이 될 겁니다. 이 식은 n에 대한 다항식이 아닙니다. n이 제곱부분에 들어가 있기 때문이죠.
갈림길은 일반적으로 보면 선택해야 한다는 것을 의미하기도 합니다. 그래서 이 예시로 넣게 되었습니다. 만약 우리가 갈림길에서 선택하지 않아도 된다면 어떨까요? 갈림길이 나오면 자신을 복제하여 보내는 겁니다. 이렇게 되면, 하나의 인간은 한 번만 들어갔다가 오면 됩니다. 다른 곳은 이미 자신의 클론이 갔다오니까요. 이렇게 되면 최대로 걸리는 시간이 겨우 2*n분이 됩니다. 이제 n에 대한 다항식이 되었습니다.

비결정 상태에서 다항시간 내에 해결 가능한 문제를 np 문제라고 합니다.
np는 비결정 상태에서만 다항시간 내에 해결 가능한게 아닙니다. 당연히 p 문제는 비결정 상태에서도 다항시간 내에 해결이 가능하죠. 그래서 p는 np의 부분[36]입니다. 바로 위 문제는 p가 아닌 np입니다. 더 엄밀하게 정의하자면 np는 답이 맞는 지 확인하는데 다항 시간 내에 가능한 문제입니다. 그 중에서 p는 답을 구하는 것까지 다항 시간 내에 가능한겁니다.
참고로 p가 아닌 np도 검산 과정은 다항 시간 내에 가능합니다. 어떤 경로가 정답이라고 할 때, 이 정답이 맞는 지 체크하려면 갔다와보면 됩니다. 그러면 2*n분이라는 다항 시간이 나오게 됩니다. 이를 이용해 다르게 정의하자면 np는 답이 맞는 지 확인하는데 다항 시간 내에 가능한 문제입니다. 그 중에서 p는 답을 구하는 것까지 다항 시간 내에 가능한겁니다.
이외에도 np 완전, np 난해라는 것이 있습니다. 그러나 일단은 여기까지 알아보도록 하겠습니다.
소인수분해는 np-p 관계를 나타낸 이미지에서 파란색 부분에 위치합니다. 바로 위 문제처럼요. 결정론적인(일반적인) 관점에서 소인수분해의 시간복잡도는 $O(2^{n/2} * 1/n)$입니다.
비결정론적인 연산이 불가능한 컴퓨터의 약점을 비대칭키는 이용합니다. 즉, p가 아닌 np문제(소인수분해)를 암호화에 이용하는 것이죠.

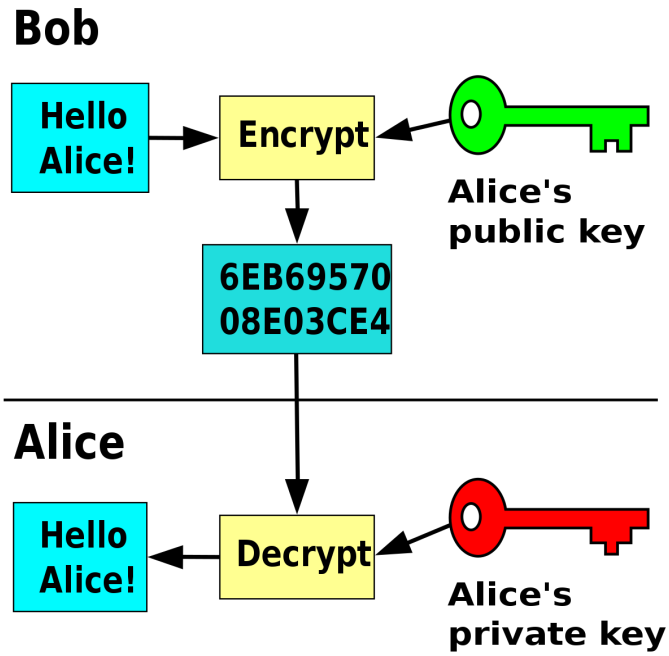


앞에서 나온 모든 대칭 키 암호화는 암호화와 복호화에 동일한 key를 사용합니다. 이는 멀리 떨어진 대상들에 대한 암호 통신을 어렵게 만듭니다. 비대칭 키 암호는 두 개의 key를

[36] 만약 모두의 예상을 깨는 증명이 나오게 되면, p랑 np는 단순히 부분 관계를 넘어 완전히 같은 것이 됩니다.

사용합니다. 하나는 모든 사람이 접근할 수 있는 공개 key와 단 한 사람만 접근할 수 있는 개인 key가 있습니다. 이 두 가지를 잘 활용하여 여러 좋은 일을 할 수 있습니다.

먼저 암호화를 진행하려면 공개 key로 평문을 암호화하면 됩니다. 이제 이 암호문은 다시 공개 key로 풀 수 없습니다. 하나로 암호화를 진행하면 반드시 사용하지 않은 다른 key로 복호화를 해야 합니다. 이렇게 되면 누구나 자유롭게 암호화를 할 수 있지만, 그것을 다시 볼 수 있는 사람은 개인 key를 가진 사람뿐이죠.



자주 사용되는 비대칭키 암호인 RSA의 작동 과정은 다음과 같습니다.

1. 두 소수 p, q를 생성합니다.
2. (p-1)(q-1)보다 작고, 서로소인 자연수 e를 구합니다.
3. (e*d) mod[37] (p-1)(q-1) = 1을 만족하도록 하는 d를 구한다.
4. N=p*q로 N을 구하고, e와 함께 공개한다. 이 둘이 공개 key이다.
5. d는 개인 key이다.
6. 연산에서 사용된 나머지 값들은 전부 버린다.(메모리에서 삭제)

이제 암호화를 진행하면 됩니다. 지금부터 설명할 것은 교과서적인 방법입니다. 실제로는 더 강한 암호를 만들기 위해 다른 방법을 사용하지만, 기본 원리에 이해는 교과서적인 방법으로도 충분합니다.
평문을 숫자로 바꾸어야 합니다. 그냥 평문의 각 문자를 아스키 코드로 바꾼 후 이 숫자들을 합칩니다. abc는 979899 이런 수가 됩니다. 이때 N보다 작아야 합니다. 이제 이 숫자를 x라고 할 때, 다음 과정으로 암호화를 합니다.

[37] mod 연산인데, 나머지 연산입니다. 3 mod 5 = 3, 10 mod 2 = 0, 10 mod 5 = 0, 10 mod 3 = 1

암호문 = $x^e \bmod N$
다시 평문을 구하려면 $(x^e \bmod N)^d \bmod N$을 합니다. 간단하게 원리를 살펴보면

$(x^e \bmod N)^d \bmod N = x^{ed} \bmod N = x^{b(p-1)(q-1)+1} \bmod N$

$= (a^{(p-1)(q-1)} \bmod N)^b * a \bmod N = 1^b * a \bmod N = a$

보다시피 공개 key로 암호화를 하면, 다시 공개 key로 풀 수 없습니다.
RSA 외에도 DH나 타원 곡선 그래프를 이용하여 암호화를 진행할 수 있습니다.

비대칭키는 서명에도 활용할 수 있습니다. 개인 key는 한 사람만 가지므로 이를 이용하는 겁니다. 어떤 메시지가 내가 보낸 것이 맞다는 것 또는 메시지를 어떤 사람이 보냈는지 판명할 때 사용할 수 있습니다.
기밀성을 유지하기 위해 사용하는 암호화랑은 다르게 서명은 위조를 방지하기 위해 사용하는 겁니다. 그래서 암호화와는 달리 애초에 메시지를 비밀로 할 필요가 없습니다. 비밀이 목적이 아니기 때문입니다.
서명은 반대로 개인 key로 암호화를 하고, 공개 key로 복호화를 합니다. 그러나 단순히 암호, 복호의 key만 바뀌었다는 것은 아닙니다.

보통의 전자서명은 메시지 전체를 개인 key로 암호화하지 않고, 해시 함수를 이용합니다. 인증, 서명하고자 하는 메시지를 암호학적 해시 함수를 이용해 해시 값을 구합니다. 그 해시 값에 대해 개인 key로 암호화를 하는 것이죠. 다른 사람들은 이미 공개된 공개 key와 메시지, 그리고 암호문을 이용해 인증을 진행하는 것입니다. 참고로 여기서 공개 key로 암호문을 복호화하는 방법은 위와 동일합니다. 위의 수식을 보면 바로 알 수 있습니다.
여기서는 간단한 원리와 작동 과정에 대해서만 알아보았습니다. 실제로는 더 복잡하고, 연산 속도를 높이기 위한 추가 알고리즘 등이 있습니다. 그러나 비정상적인 업무[38]를 수행하지 않는 웹 개발자가 여기까지 알아야할 필요는 크게 없을 것 같았습니다.

파이썬에도 간단한 rsa 라이브러리를 설치하여 사용할 수 있습니다. pip install rsa

```
import rsa

# 인스턴스 생성에서 인수로 2의 거듭 제곱을 넣어줍니다.
public_key, privily_key = rsa.newkeys(1024)
# 암호화 할 것
s = b"hello"
# 암호 생성 함수 호출
en = rsa.encrypt(s, public_key)
# 암호문 복호 함수 호출
de = rsa.decrypt(en, privily_key)
print(f'평문 {s.decode()}')
print(f'암호문 {en}')
print(f'암호문을 복호화한 것 {de}')
```

[38] 라이브러리를 직접 설계하라든지 등등.

```
import rsa

# 인스턴스 생성에서 인수로 2의 거듭 제곱을 넣어줍니다.
public_key, privily_key = rsa.newkeys(1024)
# 서명할 것
message = b'good'
signature = rsa.sign(message, privily_key, 'SHA-512')
check = rsa.verify(message, signature, public_key)
print(f'서명내용 {message}')
print(f'암호문 {signature}')
print(f'난 good을 사용한 사람이 맞다! 서명 확인 {check}')
```

- 안전한 웹

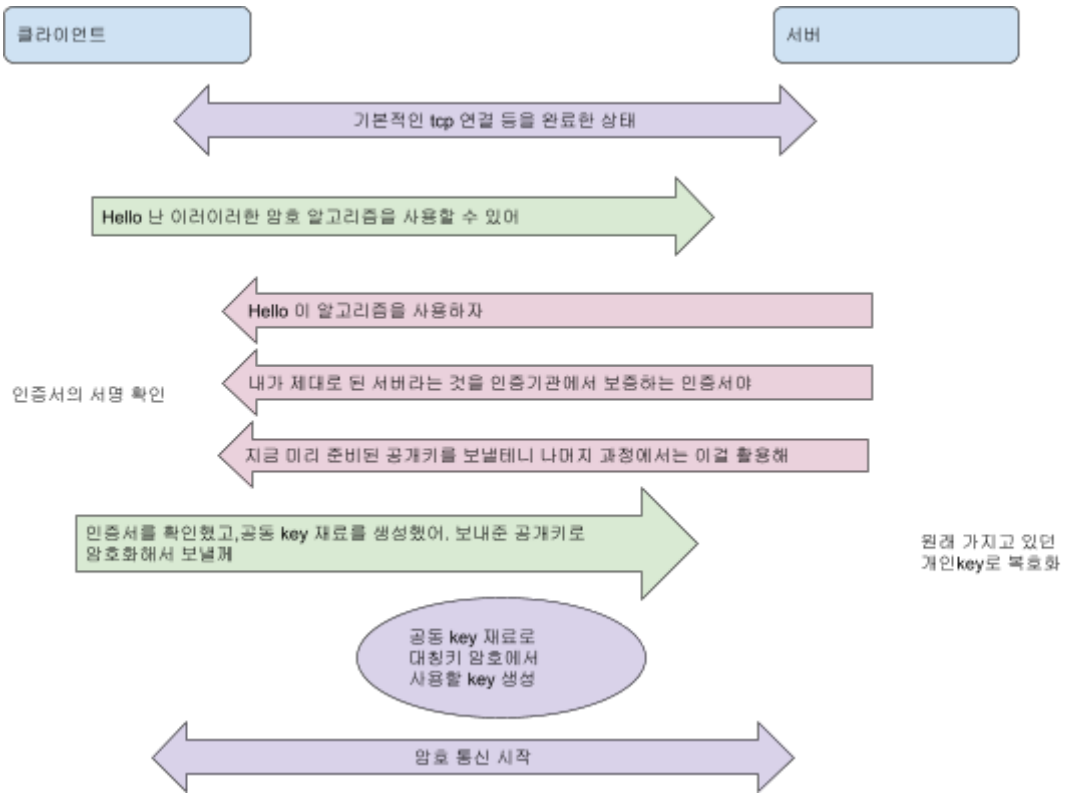
비대칭키를 이용한 암호화 과정과 서명 과정은 웹을 포함한 현대의 네트워크가 안전하게 돌아가게하는 핵심입니다. 이에 대해 구체적으로 알아보도록 하겠습니다.
예전에는 보안 소켓 계층(ssl)이라고 불렀지만, 현재는 전송 계층 보안(tls)라고 부르는 프로토콜은 인터넷 보안(기밀성, 무결성)을 책임지는 핵심입니다.

tls통해 이것들을 보장 받을 수 있습니다.

- 통신 대상자를 제외하면, 데이터 평문을 할 수 없습니다.
- 데이터가 네트워크로 이동 중 변경되지 않았다는 것을 확신할 수 있습니다.
- 누군가 여러분에게 사칭을 하거나, 여러분을 사칭하는 것이 없다는 것을 확신할 수 있습니다.

tls는 일단 http[39]에 많이 응용되기는 하지만, 사물 인터넷 등에도 충분히 적용할 수 있는 등 다양하게 사용될 수 있습니다.
현재 tls1.3까지 나왔으며, tls1.3을 사용하는 웹 사이트는 매우 안전합니다. 간단하게 tls1.3이 작동하는 과정에 대해 알아보도록 하겠습니다.
먼저 핸드셰이크를 진행해야 합니다. 클라이언트와 서버가 보안 통신을 위해 비밀 정보(key)를 공유하는 과정입니다. 원래는 과정이 엄청 길지만, tls1.3에 와서는 획기적으로 과정을 줄였습니다.

[39] 알다시피 tls로 암호화된 http를 https라고 합니다.



공동 key 재료와 이전에 보냈던 패킷의 내용들을 가지고 최종 key를 만들게 됩니다.
그리고 앞으로의 통신은 이 최종 key로 대칭키 암호화를 해서 보냅니다.[40]
참고로 인증서는 미리 지정되어 있는 기관의 까다로운 검사를 거쳐 도메인 소유자에게 발급됩니다. 인증 기관의 개인 key로 이 도메인의 소유가 확실하다는 것을 서명하고 웹 서버에게 미리 줍니다. 그리고 통신 연결 과정에서 사용자의 브라우저에 미리 내장되어 있는 공개key로 서명을 확인하는 겁니다. 웹 서비스 제공자가 인증 기관에게 까다로운 과정을 거쳐 인증서를 받는 것보다 더 까다롭게 브라우저 개발사는 인증 기관을 검토합니다. 다른 곳에다 인증 기관 개인key를 넘기지 않는 지 검토하는 것이죠. 특히 원래는 지정되었다가 퇴출당하는 경우도 많습니다. 정부기관의 요구에 쉽게 굴복하는 기관이 대표적입니다.
이 과정은 상당히 많은 부분을 생략하였지만, 전체적인 tls의 작동과정을 이해하기에는 부족하지 않습니다.
최신 버전이자 가장 안전하고, 추천하는 tls1.3은 위 과정과 좀 많이 다릅니다. 워낙 통신에서 사용할 수 있는 알고리즘이 많고 복잡하여, 그냥 위의 모델이 흔히 설명할 때 사용됩니다. 이부분에 관심이 있으신 분들이라면 RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3 을 참고해보시면 됩니다.[41]
여기에서 어떤 사이트의 tls 연결 보안 테스트를 확인해볼 수 있습니다. 여러분이 접속하는 사이트가 안전한지 확인해보세요. SSL Server Test (Powered by Qualys SSL Labs)

---

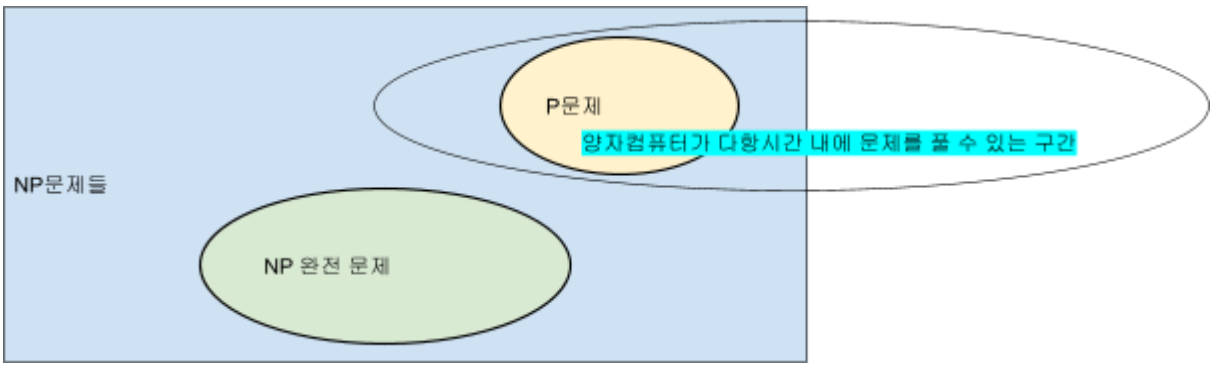
[40] 애초에 비대칭키는 대칭키의 한계를 극복하고자 나온 것입니다. 대칭키 블록 암호(aes)가 비대칭키 암호보다 빠르고, 안전합니다. 비대칭키 암호화는 구조적인 문제만 해결해주는 것이죠.
[41] 공식 문서라서 이것보다 자세하게 tls1.3을 설명하는 것이 없는 대신, 전부 영어이며 길고 복잡한 글입니다.

[42]

데이터를 안전하게 운반, 보관하기 위해서 https(tls1.3으로 암호화된 http)는 충분히 좋은 선택입니다.

- 미래의 암호

요즘 양자 컴퓨터 관련 문서들이 웹에서 많이 보입니다. 여기서는 자세하게 양자 컴퓨터의 원리를 소개하지 않겠지만 간단하게 한 문장으로 요약하자면, 비결정적인 상태가 필요한 작업에서 높은 효율을 보여줄 것으로 기대되고 개발되고 있는 차세대 컴퓨터입니다. 특히 엄청나게 높은 효율로 해결할 수 있는 문제들은 소인수분해 등이 있습니다.



웹 개발자 이것에 주목해야 하는 이유는, *tls*에서 사용되는 모든 비대칭키 알고리즘이 양자컴퓨터가 다항시간 내에 문제를 풀 수 있는 구간에 속해 있기 때문입니다.

다행인 점은 3가지가 있습니다.

- 양자컴퓨터가 갈 길은 아직 멀었습니다. 대기업부터 스타트업까지 양자컴퓨터 연구개발에 참여하고 있지만, 제대로 된 상용 수준(일반 컴퓨터 대체)까지는 멀었다고 여러 곳에서 평가하고 있습니다.

---

[42] 여러분이 이걸 보고 있으실 때는 인프런이 구형 tls버전인 tls1.1 지원을 중단하여 a등급으로 향상되었기를 바랍니다.

- 양자컴퓨터가 다항시간 내에 문제를 풀 수 없는 구간에 있는 문제를 기반으로 한 비대칭키 암호들이 개발되고 있습니다. 이를 양자 내성 암호라고 하며, NIST에서 현재 암호 알고리즘을 선발하고 있습니다. 작성일 기준으로 4라운드가 시작되었으며, 다양한 암호화, 서명 알고리즘이 경쟁하고 있습니다. Post-Quantum Cryptography PQC
- 자주 사용되는 대칭키 암호와 암호학적 해시 함수는 양자컴퓨터로도 쉽게 풀 수 없습니다. 해독하는 과정에서 효율이 아주 약간 올라가고, 이마저도 단순히 기존의 알고리즘이나 key 길이 등을 약간만 개선하면 되는 수준입니다.

위에 나온 암호 방식과 개념적으로 완전히 다른 완전 동형 암호도 개발되고 있습니다. 어떤 연산해야 하는 두 값이 있습니다. 사칙연산부터 문자열 슬라이싱 같은 것이 포함되는 겁니다. 이때 이 평문을 연산한 것과, 평문을 암호화한 뒤 연산하고 그 후 복호화를 한 것이 같은 암호가 동형암호입니다. 앞에서 나온 기존의 암호는 당연하지만 데이터를 분석, 연산하려면 아무리 중요한 정보이더라도 서버가 이를 복호해서 평문상태에서 처리해야 했습니다. 완전동형암호는 이를 극복할 수 있습니다.

모든 것이 가려지지만, 모든 연산이 가능한 암호입니다.

어떤 사람이 1 + 1을 하려고 합니다.

이 4세대 암호를 적용하고 나면, 다른 사람이 이 연산을 하는 과정을 보았을 때, 83 + 83의 연산으로 보일 겁니다.

1 + 1의 결과인 2와 83 + 83의 결과 166을 비교해보면, 1과 83을 서로 변환시키는 key가 있을 때, 2와 166에 이 key를 적용해도 된다는 겁니다.

이건 아주 기초적인 동형암호이고, 완전동형암호는 매우 높은 성능, 안정성 등을 보장하여 큰 기대를 할 수 있습니다.

사용자 맞춤 추천 알고리즘이나 개인정보 데이터 분석에서 크게 활용될 수 있습니다.

또한 현재 진행중인 완전동형암호 프로젝트는 위에서 나온 양자 내성 암호 성질을 가지고 나올 것으로 예상됩니다.

시큐어 코딩

앞에 있는 내용은 모두 이 챕터를 위한 준비 단계였다고 봐도 됩니다. 마지막인 이 챕터에서 안전하게 코드(논리적 오류나 취약점이 없는 코드)를 작성하는 법, tip 등에 대해 알아갑니다. 작성 기준은 과학기술정보통신부(한국인터넷진흥원)와 OWASP[43]에서

[43] 오픈소스 웹 애플리케이션 보안 프로젝트입니다. 가장 큰 규모의 보안 프로젝트입니다. 앞쪽에 몇 번 등장하였습니다.

공식적으로 나온 가이드라인 및 저의 개인적인 의견을 섞어서 작성한 것을 미리 알려드립니다.

1. 검색 엔진은 생각보다 꼼꼼하다.
2. 입력받은 모든 것은 격리되어야 한다.
3. 인증은 확실하게 해야 한다.
4. 블랙리스트보다는 화이트리스트
5. 웹 애플리케이션이 운영체제 명령어를 직접 사용(호출)하는 일은 없어야 한다.
6. 너(서버)가 누군지 아무도 몰라야 한다.
7. 애플리케이션 소스 코드 내에서는 반드시 하나의 정해진 문자 인(디)코딩을 사용한다.
8. 메모리를 지키자.
9. 상식적으로 행동하자.
10. 사회 공학에 대한 대비도 되어 있어야 한다.

이제 이 10가지 시큐어 코딩 원칙에 대해 자세히 알아보도록 하겠습니다.

- 상식적으로 행동하자.

상식적인 내용들이지만, 다음 내용들을 다시 한 번 상기해보도록 합시다.

개발과정에서 도움을 주는 debug는 실제 운영 환경에서는 무조건 꺼두어야 합니다. 서버 코드뿐만 아니라 데이터베이스 서버도 반드시 이를 비활성화 시켜야 합니다. 더 큰 범위로 보아서, 개발 환경과 운영 환경은 분리시키는 것이 권장됩니다.

예를 들어, 파이참을 이용해 파이썬으로 웹 서버 코드를 작성하고, 곧바로 파이참으로 이를 실행시킨 뒤 실제 운영을 하는 것이 매우 비권장된다는 겁니다. 만약 여러분이 여러분의 컴퓨터로 어떤 코드를 작성하시고, 이를 클라우드 컴퓨팅 등으로 미리 준비되어 있는 서버에 옮기고, 운영하는 구조라면 올바른 방식인 겁니다.

인증이 필요한 곳에는 꼭 인증이 있어야 합니다. 인증을 하는 과정에서 클라이언트에서 서버로 데이터를 보낼 때는 반드시 POST 방식으로 데이터를 넘기는 것이 좋습니다. 비밀번호 같이 아주 민감한 데이터는 db에 salt가 첨가된 hash값으로 저장하고, 비교할 때도 hash 값으로 비교해야 합니다. 인증은 반드시 backend에서 해야 합니다. frontend에서 하는

인증은 단순히 사용자경험[44]을 개선하는 것에 지나지 않습니다. 인증과 관련한 내용은 뒤에 매우 자세하게 다룰 겁니다.
암호 알고리즘을 사용할 때는 반드시 취약한 암호 알고리즘을 사용해서는 안됩니다. 애플리케이션뿐만 아니라 tls 연결 등에서도 1.2 아래 버전의 연결이 허용되도록 하여서는 안됩니다.

다른 외부 라이브러리나 api를 사용한다면 반드시 개발 환경에서 모든 경우를 다 테스트 해보아야 합니다. 적절한 테스트 없이 코드를 사용한다면 보안뿐만 아니라 성능면에서 좋지 못한 결과가 나올 수 있습니다. 조금이라도 수상한 api는 일단 실제 운영 환경에서 사용을 보류해야 합니다.

하드 코드는 좋은 선택이 아닙니다. 관리자 계정으로 로그인을 하는 로직을 작성할 때, 하드 코드를 하는 실수를 저지를 수 있습니다.

```
if user_id == '1234' and user_password == '1234'
```

이런 식으로 데이터가 코드 내에 직접 있는 방식의 코드 작성은 좋은 것이 아닙니다. 보안상에서 문제를 일으킬 수도 있고, 코드의 유지보수에도 좋지 않습니다. 관리자 계정으로 로그인을 할 때와 같은 경우에는 별도의 파일을 만들어 놓고, 그 파일을 열어서 그 안에 있는 암호화된 데이터와 사용자가 입력한 데이터가 맞는지 확인하도록 개발하는 것이 좋습니다.

만약 보안 사고가 발생하였다면, 확인되는 즉시 조치를 취해야 합니다. 보고체계가 있다면, 잘 숙지해두었다가 체계에 맞게 대응을 하면 됩니다. 한국인터넷진흥원에서는 보안 사고 발생시 편하게 신고할 수 있는 시스템을 갖추고 있습니다.
https://www.boho.or.kr/main.do

- 너(서버)가 누군지 아무도 몰라야 한다.

너무나 당연한 것이며, 앞에서도 여러번 이야기 했습니다. 서버의 운영체제가 무엇인지, 어떤 웹 서버 프로그램을 사용하는지, url 라우트 구조가 어떻게 되는지 당장 알았다고 해서 바로 공격하는 방법은 거의 없을 겁니다. 그러나 이는 이후에 어떤 공격이 들어올 때 매우 유용하게 사용될 수 있습니다. 일종의 약점을 미리미리 잡아두는 것이죠.
노출되어서는 안되는 서버 정보를 나름대로 중요한 순서대로 놓아봤습니다.

1. 애플리케이션 관련 보안 정보가 노출되는 것이 가장 위험합니다. 관리자 계정은 물론, 애플리케이션 내에서 사용되는 외부 라이브러리 종류 등이 있습니다. 세션 관련 정보가 노출되는 것은 바로 큰 피해를 가져올 수 있습니다.
2. 서버의 디렉토리 구조

[44] 보안 정책을 알려주어서, 사용자가 올바르게 웹 애플리케이션을 이용할 수 있도록 하는 것입니다. 사용자경험을 중요시 한다면, frontend에도 backend와 동일한 인증 로직을 넣어두면 됩니다.


디렉토리 구조는 정말 절대 노출되어서는 안됩니다.

3. 웹 프로그램의 종류와 버전

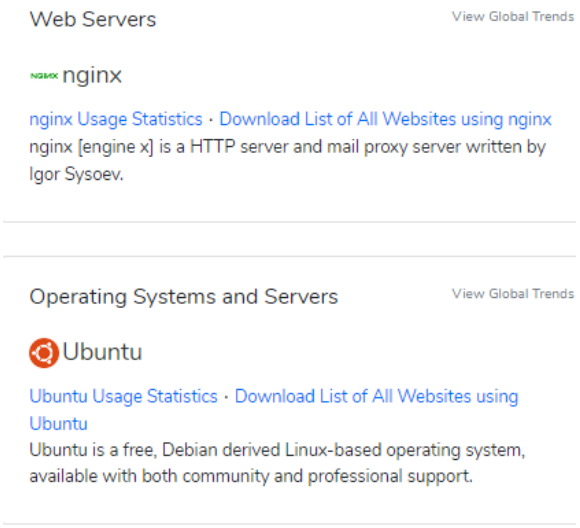
nginx나 아파치 등 웹 서버 프로그램의 정보가 노출되는 것은 좋지 않습니다. 특히 프로그램 버전이 노출되는 경우 정말 위험할 수 있습니다. 이외에도 각 프로그래밍 언어의 웹 프레임워크가 노출되는 경우도 있습니다. 이것또한 노출되는 것이 좋은 것은 아닙니다.

4. 운영체제

리눅스 우분투 18.04 이렇게 세부적으로 운영체제가 노출되는 것도 좋은 것은 아닙니다.

[BuiltWith Technology Profiler]

크롬 확장 프로그램으로 사이트를 분석할 수 있는 것이 있습니다. 이를 설치한 다음, 어떤 웹사이트에서 이 확장 프로그램을 실행시켜 보시면 놀라운 정보들을 알아낼 수 있습니다.



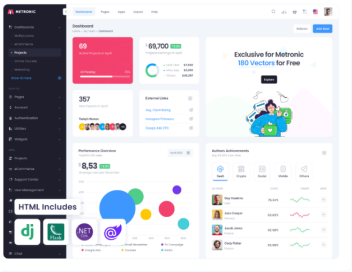
http 헤더 정보 등을 통해 개발자도 모르는 사이에 서버 관련 정보가 누출될 수 있습니다. 반드시 웹 서버 프로그램 등에서 서버 관련 정보가 노출되지 않도록 설정하는 것이 필요합니다.

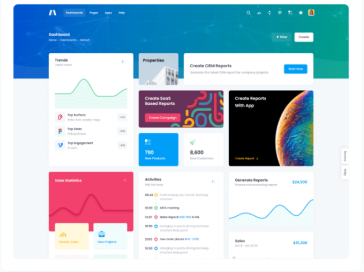
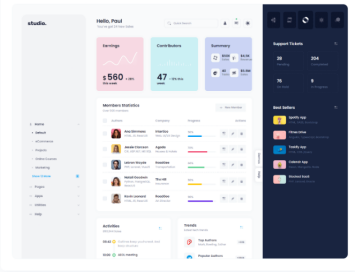
파일 업로드, 다운로드 과정에서 디렉토리 경로가 노출되지 않도록 하는 것도 필요합니다. 사용자가 올린 파일이든, 기본적으로 사용자가 접속하면 다운로드가 진행되는 파일이든, 어떤 유형이든지 파일이 이동할 때 그 과정에서 디렉토리 구조가 노출되어서는 안 됩니다.

웹 서비스는 개발자(혹은 기획자)가 정해둔 것만 해야 합니다. 너무나 당연할 듯 합니다. 그러나 보안 관련 정보가 노출되거나 하는 것은 개발을 하기 전에 미리 기획한 것이 아닐 겁니다. 모든 유형의 요청에 대해 올바른 정해진 응답**만** 해야 합니다.

비유적인 의미에서, 웹 서버는 말조심을 잘 해야 할듯 합니다.

어떤 역할을 수행하는 경로도 사용자가 쉽게 알 수 없게 해야 합니다. 당연히 중요한 경로로 접근할 경우에는 인증이 있어야 하지만, 혹시 모를 위협에 대비하여 중요한 정보를 노출하는 페이지의 경로는 쉽게 알 수 없게 하는 것이 좋습니다.



도메인/admin 같이 너무 뻔한 경로에는 관리자 페이지를 넣지 않는 것이 좋습니다. 당연히 관리자 페이지에는 엄격한 인증이 있어서 페이지에 접근해도, 쉽게 통과하지 못할 겁니다. 일반 사용자가 쉽게 관리자 페이지의 로그인 창에 접근하는 것을 원하지 않는다면 다음과 같이 복잡하게 경로를 구성할 수 있습니다. 도메인/secure/secret/admin

라우트 구조를 다음과 같이 설계하는 것도 위험할 수 있습니다.
블로그에 어떤 글을 볼려고 할 때, url이 다음과 같은 경우입니다.
/blog/1 /blog?num=1
공개적으로 글을 공개하는 경우라면 상관 없습니다. 그러나 글 중에서 공개하지 않아야 하거나 민감한 글이 있어, 외부의 접근을 쉽게 허용하지 않아야 할 경우에 이런 url 구조는 취약할 수 있습니다. 이런 상황이 필요한 경우가 직관적으로 떠오르지 않을 수 있습니다.
행사/이벤트/?신청번호=100
어떤 행사에서 진행하는 이벤트에 참가하려고 합니다. 여기서 여러분은 100번째로 신청을 하였고, 어떤 url을 주최측에서 받았습니다. 이 url로 접속하면, 경품의 당첨 유무가 나옵니다. 근데 행사에서 1번 째로 신청한 사람에게 경품을 주겠다고 합니다. 여러분은 여기에서 신청번호 파라미터를 1로 고치고 접속함으로써 악의적으로 경품을 받을 수 있습니다.
당연하지만, 공개하지 말아야 하는 글을 보기 위해서는 인증과정이 있어야 할 것입니다. 다만 앞서 관리자 페이지처럼 애초에 사용자가 공개되지 않아야 하는 페이지에 임의로 접근할 수 있다는 것 자체가 부담이 된다면, 이런 설계를 피해야 한다는 겁니다.
글 제목 hash 값 등을 사용하여 이런 구조를 피할 수 있습니다.

- 입력받은 모든 것은 격리되어야 한다.

이것또한 앞에서 여러번 강조되었던 내용입니다. 입력 받은 것이 제대로 격리되지 못한다면 injection 공격이 일어날 가능성이 커집니다.

sql에 외부 데이터를 사용할 때는 준비된 선언을 반드시 사용해야 합니다. '준비된 선언' 기능이 일종의 데이터를 격리시켜서 sql 구문을 생성하는 것이라고 해석할 수 있습니다.

cursor.execute(sql_query, (id, password))

사용자가 파일을 업로드 하는 경우에도 당연히 이 파일은 격리되어야 합니다.
하드웨어적으로 object storage를 사용하는 것도 매우 현명합니다. 소프트웨어적으로는 사용자가 올린 파일이 실행되거나, 미리 지정되어 있는 곳이 아닌 곳으로 이동되는 등 완전히 무력화가 되어 있어야 합니다. 단순히 사용자의 파일을 저장하는 용도로 기획하였다면, 반드시 서버는 파일을 아주 단순히 서버의 스토리지를 차지하는 어떤 데이터의 역할만을 가져야 할 것입니다. 파일 데이터를 분할하여 저장하는 것도 도움이 됩니다.
서버에서 저장공간을 차지하는 것, 그 이상(파일 실행)이 될 경우 문제가 될 수 있습니다. 만약 사용자가 올린 파일이 어떤 역할을 수행해야 한다면(이미지 편집, 보정 사이트 등등) 반드시 그 역할만 수행해야 하고, 그런 파일만 올라가야 합니다.(확장자 제한)
예를 들어 블로그 같은 곳이라면 사진,동영상 파일만 올리게 하는 등의 노력을 할 수 있을 것입니다.

어떤 데이터가 들어오면, 그 데이터는 반드시 미리 정해진 자신의 역할만 수행해야 합니다. 단순하게 클라이언트에서 서버쪽으로 데이터가 넘어올 때 데이터의 자료형 등이 일치되도록 넘어와야 합니다. (사용자가 쿼리를 직접 넘기는 게 좋은 건 아니지만) 만약 사용자가 직접 쿼리와 데이터를 따로 넘기는 구조라면, 쿼리 부분은 반드시 쿼리 역할만 하고, 세부 데이터 부분은 반드시 세부 데이터 역할만 해야 합니다. 사용자에게서 넘어온 대부분의 값은, 문자 자료형일 겁니다. 계속 강조하지만, 이 값은 애플리케이션 내에서 문자 자료형이어야 합니다. 애플리케이션 코드로 자료형을 바꾸는 등, 미리 계획되어 있는 변경을 제외하고, 사용자에게서 온 값이, 운영체제 명령어를 실행하거나 *DB*에서 쿼리 역할을 하는 등 문자 자료형으로서의 역할을 벗어난 것을 해서는 안된다는 겁니다.

- 인증은 확실하게 해야 한다.

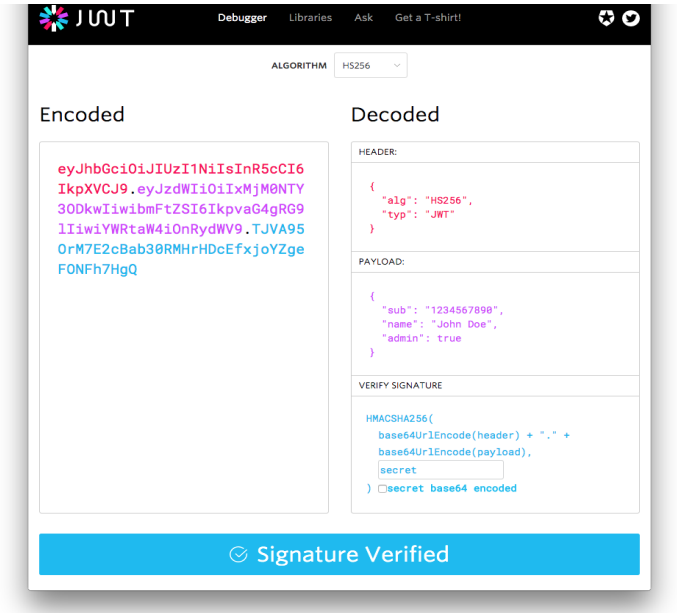
기밀성을 확실히 보장하기 위해서는 인증이 확실해야 합니다.
먼저 인증은 다른 사람이 위조할 수 있게 해서는 안됩니다. 어떤 id와 비밀번호를 모르는 사람이 id와 비밀번호를 모두 알고 있는 사람만 접근할 수 있는 곳에 접근하게 해서는 안된다는 겁니다.
예를 들어 사용자 인증을 하기 위해 쿠키를 사용한다면, 쿠키 내용이 이런 식이면 안됩니다.

로그인이 완료된 사용자

반드시 각 사용자만의 고유한 값을 이용해 쿠키의 값을 넣어야 합니다. 쿠키를 두 개를 만들어서 하나는 사용자를 구분할 수 있도록 id 평문을, 다른 하나는 이 사용자가 맞다는 것을 증명할 수 있는 비밀번호의 hash 값을 넣어두는 게 일반적입니다. 그리고 secure 설정을 해주어서 보안 강도를 좀 높여줄 수 있습니다.

세션도 쿠키를 이용한 기술이긴 하지만, 세션은 사용자의 추방이 쉽습니다. 서버에서 원하지 않는 사용자는 무조건 자동으로 로그아웃 시킬 수 있습니다. 쿠키에서는 반드시 애플리케이션이 수동으로 쿠키의 유효시간을 만료시키는 과정을 거쳐야 하지만, 세션에서는 원하는 즉시 서버에서 통제할 수 있습니다. 대신 서버에 부담이 많이 갑니다.

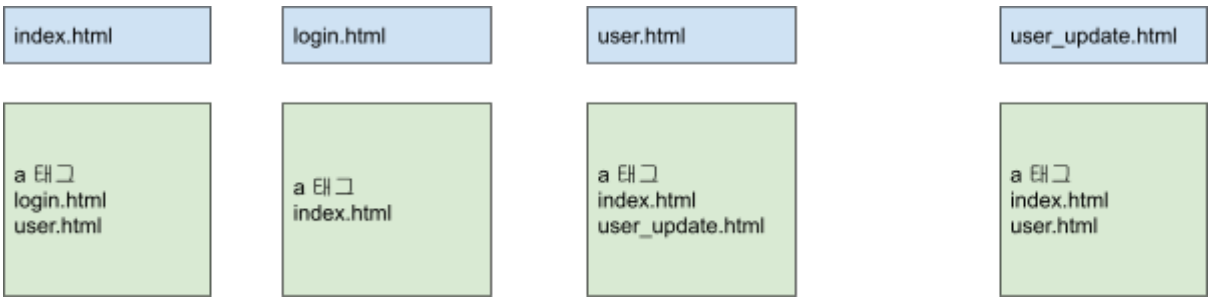
jwt는 토큰 방식입니다. 인증에 필요한 정보(사용자 id, 유효시간)를 넣어둡니다. 그리고 토큰 마지막에, 인증에 필요한 정보와 서버에서 미리 정해둔 값을 합치고 서버가 전자서명을 합니다. 인증을 확인할 때, 이 서명을 확인하는 것이죠. 서명이 확인되면, 사용자가 보낸 이 토큰을 신뢰할 수 있게 되는 겁니다. 서명만 해주고, 클라이언트에게 주면 되므로, 세션보다 서버 부담이 적고, 다른 애플리케이션에서도 사용할 수 있습니다.(소셜 로그인)



각 인증 수단은 장단점을 모두 가지고 있습니다. 중요한 점은, 애플리케이션 보안에 문제가 생겨, 사용자의 인증 수단이 탈취될 경우, 어떤 방식의 인증 수단이던 간에 정말 큰 보안 사고가 된다는 겁니다.

개발 기획 과정에서 미리 인증이 필요한 곳, 인증이 필요하다면 어떤 인증이 필요한지(단순히 로그인만 되어 있으면 되는 건지, 따로 추가 인증이 필요한지 등) 미리 명시적으로 확인하는 것이 좋습니다. 인증이 필요한 곳에는 무조건 인증을 하는 과정을 넣어야 합니다.

밑에처럼 꼼수를 사용하는 것은 좋지 않습니다.
위 그림의 상황인 것으로 가정하겠습니다.



index.html에서는 login.html과 user.html로 html태그인 a태그로 쉽게 이동할 수 있습니다.
여기서 로그인이 되어 있으면 login.html로 접근하였을 때 접근이 제한되고, 반대 상황에서는 user.html의 접근이 제한됩니다.
login.html에 정상적으로 접속한 로그인하지 않은 사용자는 다시 index.html로 돌아갈 수 있는 a태그가 있습니다.
user.html에 정상적으로 접속한 로그인한 사용자는 index.html로 돌아가거나 user_update.html로 갈 수 있는 a태그가 있습니다.

여기서 주목할 점은 반드시 로그인이 된 사용자만 user_update.html로 이동할 수 있는 a태그를 볼 수 있다는 겁니다. 몇몇 개발자는 여기서 실수를 저지를 수 있습니다. 바로 user_update.html에 접근할 때 인증하는 과정을 빼먹는 것이죠. 사용자의 페이지 이동을 도와주는 a태그가 로그인이 된 사용자만 접근할 수 있다는 것은, user_update.html이 로그인이 된 사용자만 접근할 수 있다는 의미가 아닙니다. a태그의 노출과는 별개로, 인증이 필요한 모든 페이지에는 반드시 인증 과정을 넣어야 합니다.

같은 레벨의 인증을 하는 로직은 함수나 클래스 등을 이용해, 획일화하는 것이 좋습니다. 일반 사용자로 로그인이 되어 있는지 확인하는 과정을 함수로 만들어 둡니다. 이제 일반 사용자로 로그인이 되어 있는지 확인하는 모든 과정에서 이 함수를 호출하는 겁니다. 만약 각 라우트마다 인증을 하는 코드가 다르다면, 몇몇 라우트에서 취약점이 발생할 수 있으며, 가장 중요한 점으로 이런 코드는 유지보수가 불편합니다. 굳이 보안 관련된게 아니라 단순히 성능을 개선하기 위해 코드를 바꿔야 하는 상황에서 이는 좋지 못할 겁니다.

인증을 할 때는 반드시 인증에 필요한 모든 데이터가 빠짐없이 입력되었을 때만 인증 과정을 수행해야 합니다.

아마 대부분의 경우에는 웹 애플리케이션 내에서 인증과정을 거칠 겁니다. 운영 규모가 커지면, 로드 밸런서 등이 도입되거나 각 역할에 따라 서버가 여러 개로 나누어질 겁니다. 인증 심사하는 서버와 사용자에게 응답해주는 서버는 반드시 서로가 서로를 신뢰할 수 있어야 합니다. 사용자와 직접 소통하는 서버는 반드시 신뢰할 수 있는 인증 담당 서버에게 인증 요청을 심사하도록 하고, 인증 요청을 심사하는 서버는 반드시 미리 정해진 곳에서만 들어오는 데이터만 받아야 합니다. 내부망으로만 연결되어 있다면 큰 상관이 없겠지만, 외부망으로 연결되어 있거나 물리적으로 떨어져 있는 경우에는 반드시 보안 연결을 사용하여 서로에게 요청을 보내야 합니다.

유효한 인증을 받은 사용자이더라도, 아주 중요한 작업(계정 삭제, 변경)을 할 때는 한 번 더 비밀번호 등을 입력 받아 다시 한 번 인증하도록 구성하는 것이 좋습니다. 그리고 이러한 작업을 수행하고 난 다음에는 사용자와의 별도의 연락 수단이 있는 경우(email 등) 이를 통보해주도록 합니다. 그리고 이런 중요한 작업을 너무 많이 반복하는 사용자는 모니터링 해보거나 작업의 횟수에 제한을 걸어야 합니다.

인증 과정에서 실패가 일어날 수도 있습니다. 인증 실패가 일어날 때는 사용자에게 어느정도 알려줄 필요가 있습니다. 그러나 이를 알려줄 때 세부적인 사항은 알려주지 않는 것이 좋습니다. 에러가 일어나 인증이 안된 경우에는 애플리케이션의 에러를 그대로 보여주어서는 안됩니다. id와 비밀번호가 틀린 경우에 가장 좋은 메시지는 단순히 id나 비밀번호를 잘못 입력한 것 같다는 심플한 메시지가 좋습니다.

애플리케이션뿐만 아니라 DB도 인증이 적용되어 있어야 합니다. DB도 반드시 안전하게 연결된 애플리케이션에서 온 요청에 대해서만 작업을 수행해야 합니다. 이를 달성하기 위해서는 DB에 연결을 하는 과정에서 DB에 내장된 인증 과정을 사용하는 것이 좋습니다. 많은 DB 프로그램에서 tls연결, 권한이 정해진 사용자 기능을 지원하고 있습니다.

DB 사용자 계정은 반드시 필요한 계정만 만들고, 사용자 로그인 비밀번호는 안전해야 합니다. 보안을 위해서 DB의 연결 포트를 변경하고, 애플리케이션이 DB에 접근할 때는 반드시 필요한 기능만 사용할 수 있는 권한의 사용자로서 접근을 해야 합니다.

이외에도 중요한 데이터가 오가는 통신을 할 때는 https를 이용한 암호화 통신을 사용하도록 하며[45], 회원가입을 할 때 너무 약한 비밀번호는 허용하지 않는 것이 도움이 되고, 무차별 대입 공격에 대비하기 위한 로직을 구성하는 것도 중요합니다.

- 블랙리스트보다는 화이트리스트

매우 중요한 시큐어 코딩 원칙입니다.
어떤 개발자가 db에서 어떤 기능을 사용하려고 합니다. 이 개발자는 db와 관련된 문서를 읽고, 불필요한 기능은 꺼두는 것이 좋다는 것을 발견하고, 이 기능을 제외하고 문서에서 본 나머지 기능을 꺼두었습니다. 그러나 이 문서에는 나와있지 않은 기능이 있었고, 이 기능으로 보안에 문제가 생기게 되었습니다.
이 예시에서 문제가 생긴 이유는 이 개발자는 블랙리스트 방식을 사용하였기 때문입니다. 필요하지 않은 기능을 제외하는 방식이 블랙리스트 방식입니다. 기본적으로 기능이 모두 비활성화된 상태에서 자신에게 필요한 기능만 활성화시키는 것은 화이트리스트 방식입니다.
만약 기본적으로 모든 기능이 비활성화된 상태에서, 필요한 기능만 활성시켰다면 위에 예시에서 문제가 생길 일이 없었을 겁니다.

이처럼 기본적으로 비활성화된 것에서, 필요한 것만 꺼내다 쓰는 방식은 보안에서 크게 도움이 됩니다.

사용자에게 문자를 입력받고, 거기에서 필터링을 진행해야 하는 경우 이 화이트리스트 방식이 크게 도움이 됩니다. 입력에서 제거해야 되는 몇몇 문자(특수문자 등)을 찾아내서 제거하는 것보다, 허용되는 문자의 범위를 정해두고, 여기에 포함되지 않는 문자는 모든 문자를 필터링하는 것이 좋습니다.

```
import re

string = "Hey! What's up bro?"

new_string = re.sub(r"[^a-zA-Z0-9 !?]","",string)

print(new_string)
```

[45] 진짜 도저히 https를 도입하지 못하시겠다면 임시방편으로 애플리케이션 단에서 직접 데이터에 대한 비대칭키 암호화 등을 도입할 수 있습니다. 그러나 이것은 분명히 임시방편입니다.

이렇게 파이썬에서는 re 내장 모듈을 사용해서 정규표현식을 만들고 이를 이용해 화이트리스트 필터링을 할 수 있습니다. 위의 예제에서 나온 정규표현식은 알파벳 소문자, 대문자, 숫자, 띄어쓰기, 느낌표와 물음표를 제외한 나머지 모든 것을 지워버리는 역할을 합니다.

```
import re

string = "Hey! What's up bro?"
new_string = re.sub(r"[\'\"]","",string)
print(new_string)
```

이 두 코드는 결과는 똑같을 겁니다. 그러나 바로 위에 이 코드는 블랙리스트 방식이라서 좋은 방법은 아닙니다.
의도적으로 블랙리스트를 사용해야 하는 상황이 아니라면, 보안을 위한 필터링 로직을 작성할 때는 화이트리스트 방식을 꼭 기억하시길 바랍니다.

- 웹 애플리케이션이 운영체제 명령어를 직접 사용**(**호출**)**하는 일은 없어야 한다**.**

웹 개발을 하면서 이부분에서 실수하지 않도록 주의해야 합니다.
웹 애플리케이션에서 어떤 기능을 사용하려고 할 때, 운영체제에 직접 명령어를 주어서 그 응답을 이용하는 것은 위험합니다. 아마 설치형 실행파일(exe)이나 게임 등의 서비스였다면 이 파트는 필요가 없었을 겁니다.
설마 누가 이런 짓을 하겠어? 하실 수 있지만, 크게 실수 하실 수 있습니다. **rm -rf /** 같이 어마어마한 명령을 애플리케이션에서 사용하는 것이 아닐 지라도, 큰 위협이 될 수 있습니다. 오히려 애플리케이션에서 별거 아닌 운영체제 명령을 호출한다고 더 안일하게 나올 수도 있는 것 같습니다.

어떤 서비스가 있습니다. 이 서비스에 내장된 기능 중에 사용자와의 연결상태를 체크할 수 있는 기능(icmp)이 있습니다. 개발자는 icmp를 사용하기 위해 사용자가 이 기능을 사용할 때, 사용자에게 ip 주소를 받은 후, 운영체제 명령어인 ping을 호출하여 사용합니다. 이때 만약 사용자가 이런 ip를 입력하면 어떻게 될까요?

1.1.1.1; rm -rf /

리눅스 계열은 ;을, 윈도우는 &를 사용하여 한 줄에 동시에 여러 운영체제 명령어를 사용할 수 있습니다. 이를 이용하여 서비스가 서버 운영체제의 명령어를 호출할 때 취약점을 발생시킬 수 있습니다.

필요한 작업이 있으면, 운영체제를 직접 부르는 것이 아닌, 애플리케이션 내에서 실행하는 것으로 개발해야 합니다. 위 ping 명령 같은 경우에는 다양한 라이브러리를 활용하여, 애플리케이션 내에서 패킷을 생성, 전송할 수 있습니다.
단순히 애플리케이션 내에서 운영체제 명령어를 호출하지 않는 것뿐만 아니라, injection에 의한 명령어 삽입도 조심해야 합니다.
파이썬으로 간단한 예시를 보도록 하겠습니다.

pip install ping3

```
from ping3 import *
import os
a = input()
print(ping(f'{a}'))
print(os.system(f'ping {a}'))
```

사용자에게 입력을 받고, 두 가지 방법으로 ping을 테스트하고 있습니다. 첫 번째 출력은 라이브러리를 활용하여 파이썬 코드 내에서 직접 패킷을 생성하여 ping을 테스트하고, 두 번째 출력은 운영체제 모듈을 호출하여, 직접 명령어를 사용하고, 출력을 받는 구조입니다.
코드를 실행시키고 입력을 다음을 넣어봅니다.

inflearn.com & ping google.com

라이브러리로 실행되는 곳에서 테스트를 진행할 수 없다는 false가 나오고, 운영체제 모듈을 이용하는 곳에서는 그대로 두 개의 테스트가 진행될 겁니다. 지금 입력한 것은 개발자가 의도하지 않은 명령일 겁니다. 의도하지 않은 명령을 어떻게 처리하냐에서 두 방식은 큰 차이를 보입니다.

- 검색 엔진은 생각보다 꼼꼼하다.

다양한 검색 엔진 덕분에 우리는 많은 웹 사이트를 쉽게 탐색할 수 있으며, 만든 웹 사이트를 쉽게 세상에 내놓을 수 있게 되었습니다.
그러나 너무 뛰어난 검색엔진들은 몇몇 노출되어서는 안되는 정보들을 읽고, 검색 결과에 반영합니다. 로그인을 통해서 보여지는 개개인의 정보 페이지와 관리자 페이지 등이 노출될 경우, 피해를 가져올 수 있습니다. 이를 제한하기 위해 robots.txt라는 텍스트 파일을 만들어서, 검색엔진의 수집을 통제할 수 있습니다.
도메인/robots.txt 로 접근할 경우, txt 파일이 하나 보여지면 됩니다. 안전하게 robots.txt를 보내줄려면, 도메인/robots.txt로 요청이 들어오면, 파일 자체로 바로 응답하는 것이 아닌, /robots.txt라는 요청을 backend에서 받아들이고, 응답으로 파일을 보내주는 방식을 권고합니다.(파일 업로드 취약점에서 나온 내용)

```
@app.route('/robots.txt')
def test():
```

```
    f = open('robots.txt', 'r', encoding='UTF8')
    content = f.read()
    return Response(content, mimetype='text/plain')
```

robots.txt 파일을 보내주는 것보다, 안에 있는 내용이 더 중요합니다. robots.txt 파일을 어떻게 작성해야 하는지, 작성과정에서 애플리케이션 라우트 구조가 노출되지 않도록 하는 방법에 대해 알아보겠습니다.

먼저 예시 애플리케이션의 url 디렉토리 구조는 다음과 같습니다.

도메인/ 웹 사이트 안내 내용이 있습니다.

도메인/login 로그인 페이지입니다.

도메인/signup 회원 가입 페이지입니다.

도메인/user 각 회원의 정보가 표시되는 페이지입니다.

도메인/user/update 회원 정보를 업데이트하는 페이지입니다.

도메인/user/delete 회원 정보를 삭제하는 안내 페이지입니다.

도메인/user/admin 관리자 페이지입니다.

도메인/user/hello.pdf 사용자가 올린 개인 파일입니다.

도메인/user/good.pdf 사용자가 올린 개인 파일입니다.

이제 다음 규칙에 따라 txt 파일을 작성하시면 됩니다.

모든 검색엔진에게 규칙을 적용하고 싶어요.

User-agent: *

a 검색엔진에게 규칙을 적용하고 싶어요.

User-agent: a

a 검색엔진과 b 검색엔진에 규칙을 적용하고 싶어요.

User-agent: a

User-agent: b

모두 허용하고 싶어요.

Allow: /

모두 허용하고 싶지 않아요.

Disallow: /

user페이지 그 아래 디렉토리를 허용하고 싶지 않아요.

Disallow: /user/

그냥 user로 시작 하는 모든 경로를 허용하고 싶지 않아요.

Disallow: /user

웹 사이트 안내 페이지만 허용하고 싶어요. (중첩되는 상황에서는 구체적인 것이 우선됩니다.)
Disallow: /
Allow: /$

웹 사이트 내에 있는 모든 pdf 파일에 대한 접근을 차단하고 싶어요.(pdf로 끝나는 모든 경로 차단)
Disallow: /*.pdf$

모든 검색엔진에게 탐색을 허용할 경우, txt 파일에는 다음이 들어가 있으면 됩니다.
User-agent: *
Allow: /

여러분은 a와 b 검색엔진을 마음에 들어하지 않습니다. 이 두 검색엔진은 안내 페이지만 허용하고, 나머지는 user로 시작하는 모든 경로를 제외하고 전부 허용해줍니다.
User-agent: a
User-agent: b
Disallow: /
Allow: /$

User-agent: *
Disallow: /user
Allow: /

상식적인 검색엔진은 robots.txt에 적혀있는 사항들을 준수합니다. 그래서 robots.txt를 잘 작성하는 것만으로, 검색엔진에 의해 민감한 정보가 노출되는 것을 방지할 수 있습니다. 그러나 잘못된 url 디렉토리 구조는 오히려 정보를 유출시킬 수 있습니다.
만약 관리자 페이지가 다음 경로였다면, robots.txt를 어떻게 작성했어야 할까요?
도메인/admin

이 경우 **Disallow: /admin**이 robots.txt에 적혀있어야 할 것입니다. 검색엔진에게 관리자 페이지가 잡히지 않도록 작성하는 것인데, 오히려 더 쉽게 관리자페이지의 위치가 노출될 수 있습니다. 이런 경우 때문에 아예 검색에 잡히지 않아야 하는 페이지는 한 디렉토리에 몰아 넣는 것이 좋습니다. 규칙 설명할 때 예시처럼요.

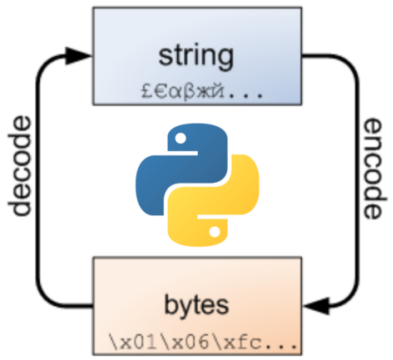
귀찮다고 검색엔진에 의한 수집을 신경쓰지 않을 경우, 수많은 사람들이 검색결과에서 여러분의 사이트가 다음과 같이 노출될 것입니다.

관리자페이지

관리자. 안녕하세요. **관리자**님. 메뉴아이콘. 교육관리 · 메뉴아이콘. 과정별 게시판 관리 · 메뉴아이콘. 콘텐츠 제공기관 **관리자**.

- 애플리케이션 소스 코드 내에서는 반드시 하나의 정해진 문자인(디)코딩을 사용한다.

상식적이면서 매우 중요한 원칙입니다.



데이터 인코딩과 디코딩에서 형식이 달라 이상한 문자가 나오는 것을 보신 경험이 있으실 겁니다.
웹 애플리케이션 내에서 잘못된 인코딩과 디코딩은 기능 상에 문제를 일으킬 수 있으며, 보안 문제도 일으킬 가능성이 있게 됩니다. 흔히 말하는 깨진 문자 등이 인코딩, 디코딩 되면서 어떠한 행동을 일으킬 수 있는 코드로 바뀌게 되고, injection이 일어날 수도 있습니다.

&lt;	<
&gt;	>
	공백
&amp;	&

&quot;	"

html에서는 태그와 관련된 특수문자를 다음 표와 같이 변환할 수 있습니다.
원래 의도는 어떤 역할을 하는 위 특수문자들을 html 내에서 콘텐츠로써 표시하기 위해 자동으로 변환되게 해두었을 겁니다. 그러나 원래 목적과 다르게 일부는 xss에서 필터링을 우회하기 위해 사용되기도 합니다.
애플리케이션 backend에서 이런 문제를 원초적으로 막기 위해서는 반드시 하나의 문자 인코딩, 디코딩을 사용하는 것입니다.

간단하게, 사용자에게 입력받은 것을 UTF8로 해석하고 DB에 저장한 경우, 사용자에게 다시 데이터를 보여주거나 애플리케이션 내에서 입력 데이터를 처리할 때, UTF8을 기준으로 처리해야 한다는 것입니다.
사용자에게 아스키 코드로 변환해서 보여준다거나, 애플리케이션 내 객체에게 데이터 처리를 맡길 때 아스키 코드로 변환해서 넘겨주거나 하는 등의 행위가 위험하다는 것입니다.

```
f = open('alpha.py', 'r', encoding='UTF8')
a = f.read()
ff = open('testing.py', 'w', encoding='UTF8')
ff.write(a)
```

위처럼 utf8으로 파이썬 파일을 읽고, 새 파이썬 파일에 utf8으로 입력을 해주었습니다.
참고로 대부분의 프로그래밍 언어 파일이나 일반적으로 작성하는 텍스트 파일은 보통 utf8으로 되어 있습니다.

```
f = open('alpha.py', 'r', encoding='UTF8')
a = f.read()
ff = open('testing.py', 'w', encoding='UTF32')
ff.write(a)
```

이런 식의 코드는 일관되지 못합니다. testing.py를 읽을 때, 인코딩 형식을 alpha.py와 다른 utf32로 읽어야 정상적으로 읽히기 때문입니다. 예제 코드와 같은 과정은 간단한 과정이며, 문제가 생겨도 간단한 파일 에러가 생길 뿐이지만, 애플리케이션 코드가 커지고, 분산될수록 인코딩 디코딩 형식을 지키는 것은 보안면에서 중요해집니다.

- 메모리를 지키자.

낮은 성능의 컴퓨터를 사용하시면서 흔히 무겁다고 표현하는 프로그램들을 동시에 여러 개를 실행하게 되면, 갑자기 느려지는 경험을 겪어본 적이 있으실 겁니다. 이는 보통 메모리가 부족하여 발생하는 일입니다.

DDOS 공격은 한정된 네트워크 트래픽을 이용하는 공격 외에도 서버의 메모리를 빠르게 사용해나가며 공격하는 유형도 있습니다. 애플리케이션 DDOS는 큰 규모의 공격이 아니더라도, 개발자의 잘못된 설계로 인해, 큰 피해를 가져올 수 있습니다.
굳이 공격이 일어나지 않더라도, 메모리 누수가 생기지 않는 지 검토하는 것은 성능상에 큰 도움이 됩니다.

즉, 부적절한 자원 해제는 공격자가 쉽게 DOS 공격을 할 수 있게 만들어 줍니다. 유한한 하드웨어 자원을 할당 받아 사용한 뒤, 더 이상 사용하지 않을 경우에는 잘 반환해줘야 합니다. 그러나 잘못된 설계로 제대로 반환이 되지 않는 경우가 생길 수 있습니다.

이 챕터에서의 메모리 관리는 쓰레기 수집[46]과 같은 내용보다는 파일을 열고, 전송하고, 닫을 때, 이 과정을 안전하게 처리해야 한다는 내용 위주입니다.
가장 일반적인 예시로 파이썬으로 간단한 파일을 여는 것을 생각해보겠습니다.

```
try:
    f = open('list.txt', 'r')
    text = f.read()
    f.close()
except Exception as e:
    print(e)
```

예외처리 문이 있어 이 코드는 안전하게 보일 수 있습니다. 그러나 아래 표시된 곳에서 문제가 발생하면 어떤 상황이 발생할까요?

```
try:
    f = open('list.txt', 'r')
    text = f.read()
    f.close()
except Exception as e:
    print(e)
```

[46] 애플리케이션에서 사용하는 메모리를 하나하나 삭제하는 과정을 반복할 필요가 없게 만들어 줍니다. 파이썬, 자바 계열 모든 언어에 탑재되어 있습니다.

파일이 열리지만, 파일이 닫히는 과정은 수행되지 않습니다. 이렇게 되면, open()을 위해 할당된 메모리 자원이 제대로 반환되지 않는 상황이 만들어 질 수 있습니다.

파이썬에서는 with 문을 사용하여, 아주 간편하게 이 문제를 해결할 수 있습니다.

```
with open('list.txt', 'r') as f:
    text = f.read()
```

with 문은 할당된 자원을 자동적으로 잘 반환해줄 수 있습니다.

파이썬에서 이 명령은 대략적으로 같은 의미를 지닌 다음 코드로 변환할 수 있습니다. with 문과 같은 것이 없는 언어에서 사용할 수 있습니다.

```
def fun():
    f = open('list.txt', 'r')
    try:
        text = f.read()
    except Exception as e:
        print(e)
    finally:
        f.close()
        return
```

어떤 과정을 하던지, 반드시 받은 자원은 돌려줘야 합니다.

번외로, 웹 서버(리눅스)를 운용하시다 보면, 메모리 상태를 조회할 때, 캐시가 많아서 사용가능한 메모리가 적은 것을 보실 수 있습니다. 캐시는 속도가 느린 디스크에 최대한 덜 접근하기 위해 사용하는 것입니다. 그 중 페이지 캐시는 읽은 파일을 완전히 반환하지 않고 메모리에 계속 보관하는 것입니다.

free 명령을 통해 현재 캐시를 확인합니다. 679024만큼 입니다.

```
              total        used        free      shared  buff/cache   available
Mem:        1008636      142692      186920         152      679024      711620
Swap:        222000       37632      184368
```

간단하게 아무 파일이나 cat으로 열어주고, 다시 free 명령으로 확인해봅니다.

```
              total        used        free      shared  buff/cache   available
Mem:        1008636      142752      186580         152      679304      711540
Swap:        222000       37632      184368
```

캐시가 약간 늘었습니다.

당연하지만 사용가능한 메모리가 부족해지면, 캐시는 우선적으로 반환됩니다. 그래서 캐시로 인해 메모리가 부족해질 걱정을 하실 필요는 없습니다. 그럼에도 캐시가 쌓여가는 것이 불편하신 분들은 다음 명령어로 캐시를 지워줄 수 있습니다.

[How to Clear RAM Memory Cache, Buffer and Swap Space on Linux]

echo 1 > /proc/sys/vm/drop_caches

echo 1(페이지 캐시)뿐만 아니라 2와 3(모든 캐시)도 있지만, 2와 3을 지우는 것은 신중해야 합니다. 캐시를 비워서 여유있는 메모리 공간을 확보하여 안정성을 높일 수 있지만, 너무 자주 캐시를 비우는 것은 큰 속도 저하를 일으킬 수 있다는 것을 잊지 마시길 바랍니다.

- 사회 공학에 대한 대비도 되어 있어야 한다.

어쩌면 가장 중요할 수도 있습니다. 지금까지는 기술적인 부분만 다뤄왔습니다. 이 파트는 기술과 큰 관련이 없지만, 올바른 보안을 위해 꼭 명심해두어야 합니다.
사회 공학을 이용한 공격은, 사람을 속여 컴퓨터 시스템에 침입하는 등의 공격을 의미합니다. 아무리 기술적으로 철저한 보안 시스템이라도 이를 쉽게 무력화 시킬 수 있습니다.
서비스 업체가 방화벽 등과 같은 각종 보안 기술에 아무리 많은 써도, 서비스 관리자 중에서 공격자를 도와준다면, 보안을 위해 업체가 사용한 돈은 모두 쓸모없는 것이 될 것입니다.
사회 공학 공격을 자주 당하는 유형은 다음과 같습니다.

- 보안에 신경쓰지 않는 경우가 대표적일 겁니다. 회사 자체가 보안 교육에 신경[47]을 안 쓰거나, 개개인이 보안의 중요성을 몰라 큰 신경을 안 쓰는 경우가 있습니다.
- 높은 권한, 특별한 권한을 가진 사람은 어쩔 수 없이 사회 공학 공격에 쉽게 자주 노출됩니다. 민감한 정보에 접근하거나 변경할 수 있는 사람은 공격자가 크게 관심을 가지는 표적입니다.
- 어떤 웹 서비스 회사 본사 사옥 경비원 등, 웹 서비스와 직접 관련이 없는 사람도 포함될 수 있습니다. 자신도 모르게 정보를 유출할 수 있습니다.

현실에서의 인간-인간 관계에서 사회 공학 공격이 많이 일어납니다. 그러나 사회 공학의 정의를 생각해보면, 꼭 현실에서의 인간-인간 관계일 필요는 없습니다. csrf 공격이나, 피싱 사이트 링크가 포함된 문자를 보내서 사용자가 위조된 사이트에 접속되게 하여, id와 pw를 알아내는 것도 사회 공학에 포함됩니다. 이 부분은, 앞에서 나온 인증을 확실하게 함으로써 예방할 수 있습니다.
사회 공학에는 고도의 심리학 기법이 사용됩니다. 대화 과정에서 자연스럽게 정보 유출을 유도하는 기법 등이 있습니다. 물론 이런 거창한 방법 말고도 비밀번호를 입력할 때 몰래 힐끗힐끗 키보드 자판을 보는 간단한 행위도 공격입니다.

[47] 기술적인 보안은 챙기면서 사회 공학을 챙기지 않는 것은 멍청하다고 할 수 있을 겁니다.

사회 공학 공격을 방어하기 위해서는 개발자를 포함한 웹 서비스 관련 모든 사람이 다음 사항을 숙지하고 있어야 합니다.

- 개발, 운영 관련 사항은 반드시 단호하게 나와야 합니다. 외부인(개인적인 친분 포함)은 물론 내부자(상급자)에게도 매우 간단한 정보를 제공하는 것도 검토를 충분히 해야 합니다. 정보가 반드시 필요한 사람인지, 그 사람을 확실하게 신뢰할 수 있는지, 그 정보를 이용하는 작업을 본인이 대신할 수 있는지(정보가 최대한 확산되지 않도록 하기 위해) 등을 검토해야 합니다. 이를 위해 개개인은 보안 의식을 잘 내재하고 있어야 합니다.
- 개발, 운영용 컴퓨터 등으로 완전히 신뢰하기 어려운 출처에서 온 모든 외부 파일을 열 때는 항상 경계해야 합니다.
- 개발, 운영용 컴퓨터 등으로 외부 링크에 접속할 때는 반드시 해당하는 링크가 맞는지 꼭 확인합니다. 보통의 공격자는 dns 서버나 웹 서버를 직접 공격하기 보다는, 대충 비슷한 사이트로 피싱하는 경우가 많습니다. 만약 외부 사이트에서 인프런으로 연결해준다면, 반드시 접속하고 난 다음 inflearn.com이 맞는 지 확인합니다. 텍스트는 inflearn.com이나 실제로 연결된 하이퍼링크는 피싱사이트일 수 있습니다.