Učebný text k prednáške
Bezpečné programovanie 1

Richard Ostertág, KI FMFI UK Bratislava 2020

Materiál je výstupom Rozvojového projektu Univerzity Komenského a Ministerstva školstva, vedy, výskumu a športu SR č. 002UK-2-1/2018 – „Vzdelávanie pre informačnú spoločnosť“ v oblasti Podpora vysokých škôl pri plnení záväzkov prijatých v rámci Národnej koalície pre digitálne zručnosti a povolania SR.

Jazyková úprava:

Rukopis neprešiel jazykovou úpravou.

Obsah

Obsah i

Zoznam obrázkov ii

1 Úvod 1

2 Bezpečné programovanie 1 3
2.1 Verejné databázy zraniteľností . . . 3
2.1.1 MITRE CVE . . . 3
2.1.1.1 MITRE CWE . . . 4
2.1.2 NIST NVD . . . 5
2.2 Secure by Design, by Default, in Deployment . . . 7
2.3 Analýza kódu . . . 7
2.3.1 Statická analýza kódu . . . 7
2.3.2 Dynamická analýza kódu . . . 9
2.4 Znalosť programovacieho jazyka . . . 10
2.5 Znalosť operačného systému . . . 10
2.6 Allow vs. deny list . . . 10
2.7 Zraniteľnosti vo formátovacom reťazci . . . 11
2.7.1 C . . . 11
2.7.2 Perl . . . 12
2.7.3 PHP . . . 13
2.7.4 Python . . . 13
2.8 Vyčerpanie zdrojov . . . 14
2.8.1 Neobozretné protokoly alebo algoritmy . . . 15
2.8.2 Útoky na algoritmickú zložitosť . . . 16
2.8.2.1 ReDoS . . . 16
2.9 Chybná správa pamäte . . . 17

3 Záver 21

Bibliografia 23

Zoznam obrázkov

2.1 Vyhľadanie zraniteľností na MITRE CVE pre „Microsoft Office“ 3
2.2 Podrobnosti k zraniteľnosti „CVE-2020-16957“ 4
2.3 Podrobnosti k zraniteľnosti „CWE-787: Out-of-bounds Write“ 6
2.4 Záznam k zraniteľnosti „CVE-2020-16957“ v NVD 19

Kapitola 1

Úvod

Pri vývoji aplikácií sa kladie veľký dôraz na funkcionalitu výsledného produktu a jeho cenu, ale aj na rýchlosť samotného vývoja. Častokrát sa však pritom „zabúda“ na jeho bezpečnosť. Či už je to neznalosť (ktorá ale neospravedlňuje) alebo úmyselné zanedbanie tejto problematiky (napríklad z dôvodu časovej alebo finančnej tiesne), tak výsledkom je „nebezpečný“ produkt. Správne by sme mali napísať „nie bezpečný“, ale je potrebné si uvedomiť, že pokiaľ nastane zneužitie niektorej zo zraniteľností[1], tak naozaj môže nastať nebezpečná situácia, finančná škoda, strata dobrej povesti alebo aj ujma na zdraví. Podobne aj schody, ktoré nemajú zábradlie považujeme za nebezpečné. Samozrejme, pokiaľ sa nezapotácame, môžeme ich bez následkov roky používať a byť ich spokojným používateľom. Podobne ako aplikácia, ktorá nie je bezpečná, môže roky vynikajúco plniť potreby jej používateľa. Jedného dňa však môže prísť k zneužitiu niektorej jej zraniteľnosti. Následky takéhoto útoku môžu byť veľmi rozmanité, napríklad:

- pozmenenie, vloženie alebo vymazanie údajov v aplikácii
  - prevod peňazí na iný účet alebo v inej sume,
  - navýšenie počtu objednaných kusov po zaplatení objednávky, …
- „zhodenie“ alebo preťaženie aplikácie
  - časté pády pokladničného systému môžu viesť k dlhým radom, nespokojným zákazníkom a strate tržby a dobrej povesti
  - preťaženie aplikácie spomalí jej odozvu a pokiaľ ide o riadiaci systém, tak môže prísť napríklad k zrážke a tak aj škodám na zdraví, …
- nekorektné výsledky alebo závery aplikácie
  - pozmenením spracovaných údajov môže prísť ku schváleniu pôžičky, ktorá by inak neprešla, …

[1] ktoré sa v takejto aplikácii s veľkou pravdepodobnosťou nachádzajú

- obídenie kontroly prístupu a získanie neoprávneného prístupu
  - obídením kontrol môže neplatiaci útočník získať prístup k plateným článkom …
- vykonanie ľubovoľného programu na systéme, kde je aplikácia prevádzkovaná
  - cez nainštalované aplikácie môže útočiť nielen na prvotne napadnutú aplikáciu, ale získaný prístup využiť aj na útok na iné časti systému, ktoré pôvodne pre neho neboli prístupné …

Z vyššie uvedeného jasne vyplýva, že je dôležité venovať náležitú pozornosť bezpečnosti počas celého životného cyklu aplikácie. V tomto materiáli sa sústredíme na bezpečnostné zraniteľnosti, ktoré môžu byť do aplikácie zavedené nielen z dôvodu zlého návrhu ale aj rôznych „temných častí“ samotných použitých programovacích jazykov, či algoritmov.

Pri písaní tohoto materiálu sme vychádzali okrem vlastných skúseností aj zo zdrojov [1–10], ktoré odporúčame aj čitateľovi, ktorý ma záujem o hlbšie samoštúdium.

Kapitola 2

Bezpečné programovanie 1

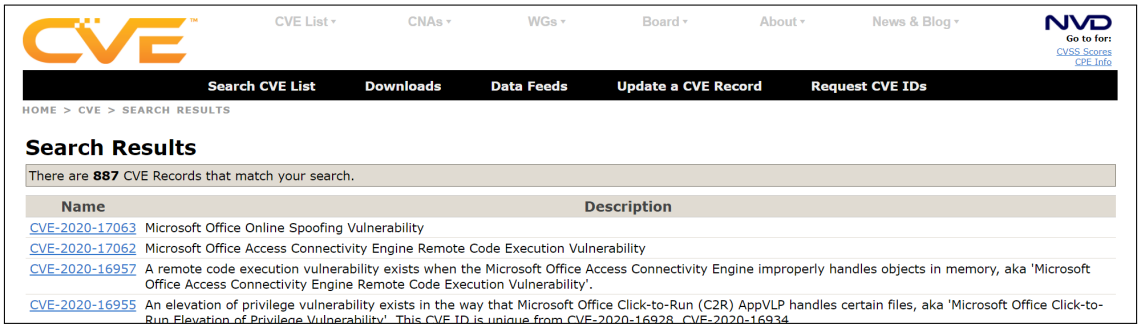
Vývoj bezpečných aplikácií nie je triviálna záležitosť. Ako dobrý dôkaz tejto skutočnosti možno uviesť rozsiahly obsah databáz bezpečnostných zraniteľností.

2.1 Verejné databázy zraniteľností

Existuje viacero databáz zraniteľností. Niektoré z nich sú verejne dostupné. My si v nasledovných podčastiach spomenie asi dve najznámejšie MITRE CVE a NIST NVD.

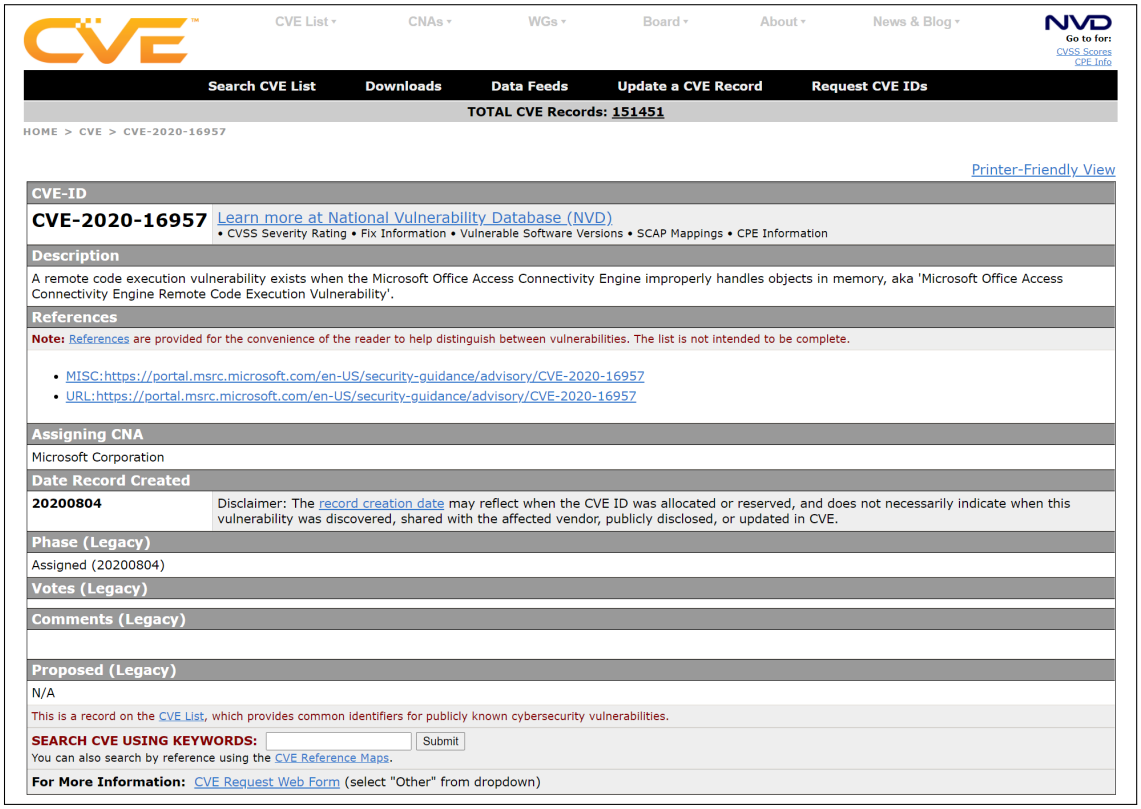
2.1.1 MITRE CVE

Poslaním programu CVE (`http://cve.mitre.org/`), ktorý je prevádzkovaný spoločnosťou MITRE, je identifikovať, definovať a katalogizovať verejne dostupné zraniteľnosti v oblasti kybernetickej bezpečnosti. Skratka CVE pochádza z „Common Vulnerabilities and Exposures“ alebo „Common Vulnerabilities Enumeration“. Na internetovej adrese `https://cve.mitre.org/cve/search_cve_list.html` môžeme vyhľadať zraniteľnosti vyhovujúce zadaným podmienkam. Túto možnosť môžeme využiť napríklad na vyhľadanie zraniteľností v danom produkte:



Obr. 2.1: Vyhľadanie zraniteľností na MITRE CVE pre „Microsoft Office“

Ako vidno na obrázku vyššie, tak CVE eviduje od svojho začiatku v roku 1999 do roku 2020 presne 887 zraniteľností. Keď si jednu z nich vyberieme, napríklad CVE-2020-16957, tak sa dostaneme na stránku:



Obr. 2.2: Podrobnosti k zraniteľnosti „CVE-2020-16957“

Na stránke opisujúcej konkrétnu zraniteľnosť nájdeme základné informácie ako CVE-ID, ktoré zraniteľnosť jednoznačne identifikuje, linku na záznam v NIST NVD (pozri podčasť 2.1.2), popis zraniteľnosti, referencie na ďalšie zdroje a informácie k zraniteľnosti.

2.1.1.1 MITRE CWE

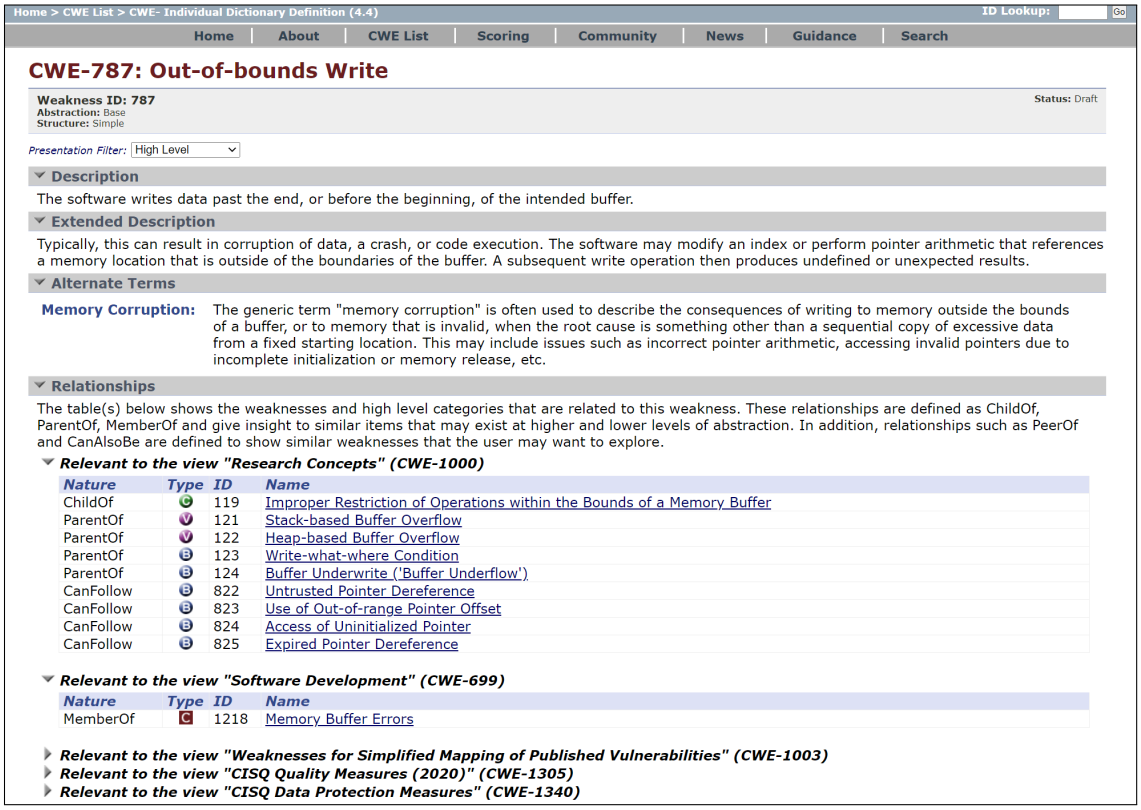
CWE (`http://cwe.mitre.org/`) je komunitou vyvinutý a udržiavaný komplexný slovník softvérových a hardvérových zraniteľností (Common Weakness Enumeration). Obsahuje stovky (skoro až tisícku) všeobecne popísaných zraniteľností kategorizovaných podľa rôznych pohľadov. Celý zoznam môžeme nájsť na `http://cwe.mitre.org/data/slices/2000.html`. 25 najnebezpečnejších softvérových chýb v roku 2020 nájdeme na `http://cwe.mitre.org/top25/`:

#	ID	Názov	Skóre
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46,82
2	CWE-787	Out-of-bounds Write	46,17
3	CWE-20	Improper Input Validation	33,47
4	CWE-125	Out-of-bounds Read	26,50
5	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23,73
6	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20,69
7	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19,16
8	CWE-416	Use After Free	18,87
9	CWE-352	Cross-Site Request Forgery (CSRF)	17,29
10	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16,44
11	CWE-190	Integer Overflow or Wraparound	15,81
12	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13,67
13	CWE-476	NULL Pointer Dereference	8,35
14	CWE-287	Improper Authentication	8,17
15	CWE-434	Unrestricted Upload of File with Dangerous Type	7,38
16	CWE-732	Incorrect Permission Assignment for Critical Resource	6,95
17	CWE-94	Improper Control of Generation of Code ('Code Injection')	6,53
18	CWE-522	Insufficiently Protected Credentials	5,49
19	CWE-611	Improper Restriction of XML External Entity Reference	5,33
20	CWE-798	Use of Hard-coded Credentials	5,19
21	CWE-502	Deserialization of Untrusted Data	4,93
22	CWE-269	Improper Privilege Management	4,87
23	CWE-400	Uncontrolled Resource Consumption	4,14
24	CWE-306	Missing Authentication for Critical Function	3,85
25	CWE-862	Missing Authorization	3,77

Ku každej zraniteľnosti nájdeme v katalógu jej podrobný popis spolu s kategorizáciou zraniteľnosti. Ukážku takéhoto popisu pre zraniteľnosť „zápis mimo rozsah“ vidíme na obrázku 2.3 na strane 6.

2.1.2 NIST NVD

NVD (https://nvd.nist.gov/) je americké vládne úložisko údajov o zraniteľnostiach (National Vulnerability Database). Je založené na CVE. Okrem toho obsahuje aj kontrolné zoznamy pre zabezpečenie (security checklists), názvy produktov, ako aj metriky pre meranie



Obr. 2.3: Podrobnosti k zraniteľnosti „CWE-787: Out-of-bounds Write“

dopadu (závažnosti) zraniteľností. Tieto podrobnosti môžeme vidieť na obrázku 2.4 na strane 19. Tu sa môžeme dozvedieť, že ide o zraniteľnosť s vysokým skóre 7,8 (na škále od 0 do 10).

Okrem toho je každá zraniteľnosť popísaná aj „Common Vulnerability Scoring System“ (CVSS) vektorom. V tom prípade má vektor nasledovnú podobu:

```
CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H
```

Význam jednotlivých položiek je nasledovný:

CVSS:3.1 – použitá verzia CVSS

AV:L – Attack Vector: Local

AC:L – Attack Complexity: Low

PR:N – Privileges Required: None

UI:R – User Interaction: Required

S:U – Scope: Unchanged

C:H – Confidentiality Impact: High

I:H – Integrity Impact: High

A:H – Availability Impact: High

K zraniteľnostiam nájdeme aj ďalšie informácie, ako napríklad odkaz na odporúčania a záplaty od výrobcu zraniteľnej aplikácie a zoznam verzií, ktorých sa zraniteľnosť týka.

2.2 Secure by Design, by Default, in Deployment

Aplikácie/systémy by mali byť od začiatku navrhované s ohľadom na ich bezpečnosť. Vývojár by mal najlepšie vedieť posúdiť, aké možnosti/nastavenia sú pre aplikáciu potenciálne „nebezpečné“. Všetky takého „nebezpečné“ nastavenia by mali mať predvolené bezpečné hodnoty a ich nastavenie by malo byť zdokumentované v návode na použitie aplikácie. Pretože sa nedá predpokladať, že by zákazník poznal systém lepšie, tak by mal inštalačný a konfiguračný program hneď po inštalácii použiť tieto bezpečné predvolené hodnoty. Ak existujú výnimky z tohto pravidla, tak by sa mali pri spustení aplikácie zobrazovať zodpovedajúce varovania (či už na obrazovku alebo do logov/záznamov).

Používatelia často nasadzujú aplikáciu čo najrýchlejším odklikaním inštalačného programu. S konfiguráciou sa „trápia“ len dovtedy, kým sa im nepodarí sprevádzkovať potrebnú funkcionalitu aplikácie. Často sa potom zabúda na dokonfigurovanie bezpečnostných nastavení, ako napríklad zmena štandardných hesiel, nastavenie prístupových práv a podobne.

Používateľsky najprívetivejšia je situácia, keď inštalačný program nastaví aplikáciu čo najvoľnejšie, aby bola čo najväčšia pravdepodobnosť, že všetko bude hneď správne pracovať. Dovolíme si však tvrdiť, že prísne východzie bezpečnostné nastavenie je lepšie. Donúti používateľa nakonfigurovať systém tak, aby fungoval, a ak treba pritom zmeniť nejaké bezpečnostné nastavenia, má možnosť sa s nimi oboznámiť a správne ich nastaviť. Pri opačnom prístupe by benevolentné počiatočné nastavenie ostalo nepovšimnuté (pretože by systém hneď pracoval) a potenciálne bezpečnostné hrozby by ostali skryté až do ich zneužitia.

2.3 Analýza kódu

Pri vývoji softvéru existujú rôzne možnosti, ako odhaliť chyby skôr ako útočník. Jednou takou možnosťou sú statické kontroly, druhou napríklad dynamické kontroly. Tieto dve kategórie si popíšeme v nasledovných podčastiach.

2.3.1 Statická analýza kódu

Statická analýza kódu prebieha bez jeho spustenia. Môže to byť napríklad počas kompilácie zdrojových kódov. Kompilátory kontrolujú hlavne také chyby, ktoré znemožňujú generovanie výsledného strojového kódu. Ide teda hlavne o syntaktické chyby a časť sémantických

chýb. Avšak program, ktorý „vyzerá“ podozrivo alebo „nedáva zmysel“ nemusí byť pre kompilátor ani dôvodom pre zobrazenie varovania[1]. Z tohto dôvodu vznikli rôzne ďalšie nástroje, ktoré staticky analyzujú zdrojový kód a upozorňujú na podozrivé miesta[2].

Medzi takéto nástroje sa radia aplikácie pre statickú kontrolu sémantických chýb. Anglicky sa často nazývajú „linter“ (napríklad ESLint pre JavaScript, hlint pre Haskell, clang-tidy[3] pre C++, atď.). Dokážu identifikovať klasické problémy, ako sú napríklad nepoužitie deklarovanej premennej (nemusia nájsť všetky), použitie premennej bez jej inicializácie (môžu nájsť niečo navyše). Obvykle je detekcia týchto problémov nerozhodnuteľný alebo NP-úplný problém. Preto tieto nástroje robia iba tzv. konzervatívnu aproximáciu. Ukážeme si takúto situáciu na nasledovnom príklad (program je v jazyku Python):

```
1  if p():
2    x = 0
3  else:
4    y = 0
5  if q():
6    y = 1
7  else:
8    x = 1
9
10 a = x
```

Je problém rozhodnúť, či na riadku 11 je premenná x inicializovaná alebo nie. Pokiaľ je predikát p pravdivý, tak sa inicializuje na riadku 2. Ak je pravdivý predikát q, tak sa inicializuje na riadku 9. Na riadku 11 nezáleží na tom, kde sa premenná x inicializovala. Potrebujeme však mať istotu, že sa tak stalo. Inak povedané, potrebovali by sme dokázať, že formula $p \vee \neg q$ je splnená za každých okolností (čo je napr. ekvivalentné tomu, že $q \implies p$). Vieme si predstaviť, že nie je ťažké zvoliť p a q tak, aby bol tento dôkaz mimo možnosti použitého nástroja. V takomto prípade nástroj radšej namiesto presného výsledku upozorní na možnosť, že premenná nemusí byť inicializovaná, aj keď sa to v skutočnosti nemusí nikdy stať.

Okrem aplikácií, ktoré kontrolujú sémantickú stránku programu, sa často využívajú aj aplikácie, ktoré kontrolujú (prípadne rovno upravujú) použité formátovanie kódu, tzv. štýl. Takéto aplikácie sa často volajú „code style checker“ alebo „beautifier“ (ak kód aj preformátujú). Napríklad Checkstyle pre Javu, CStyle pre C/C++ alebo js-beautify pre JavaScript.

Dodržiavanie jednotného štýlu medzi rôznymi programátormi pracujúcimi na tom istom projekte je dôležité pre zachovanie prehľadnosti a jednoznačnosti zapísaného kódu. Čitateľný a správne pochopený kód potom vedie k menšiemu počtu chýb a prípadných zraniteľností.

Nie všetky postupy zo statickej analýzy kódu musia byť v podobe nejakej aplikácie. Zaraďujeme sem aj napríklad vzájomnú kontrolu návrhu alebo kódu (design / code review).

[1] Táto situácia sa dnes už výrazne zlepšila. Mnohé kompilátory zobrazujú v takýchto situáciách varovania.

[2] Aj keď kompilátory dnes robia časť práce týchto nástrojov, stále dokáže špecializovaný nástroj identifikovať viac potencionálnych problémov.

[3] Ako vidieť z tohto príkladu, nemusí mať každý linter v názve lint.

V tomto prípade ide o klasickú kontrolu viacerých očí. Jeden programátor, ktorý opakovane číta sám po sebe svoj kód, má tendenciu vidieť napísané to, čo chcel a nie to, čo je v programe naozaj implementované. Vzájomná kontrola kódu pomáha aj lepšiemu pochopeniu kódu ostatnými členmi tímu, čo vedie k lepšej zastupiteľnosti.

2.3.2 Dynamická analýza kódu

Keďže statická kontrola nedokáže spoľahlivo odhaliť všetky chyby v programe, tak sa častokrát využíva aj dynamická kontrola, teda kontrola počas behu programu.

Typickým príkladom dynamickej kontroly je kontrola, či všetka alokovaná pamäť bola nakoniec aj uvoľnená. Takúto kontrolu robí napríklad nástroj Memcheck od firmy Valgrind. Všetky čítania a zápisy do pamäte sú týmto nástrojom za behu kontrolované a volania `malloc`, `new`, `free`, `delete` sú zachytené. Vďaka tomu Memcheck dokáže zistiť, či program:

- nepristupuje k pamäti, ku ktorej by nemal (ako oblasti, ktoré ešte neboli pridelené, oblasti, ktoré boli uvoľnené, oblasti za koncom haldy, neprístupné oblasti zásobníka),
- nepoužíva neinicializované hodnoty nebezpečným spôsobom,
- nezabúda uvoľňovať alokovanú pamäť (memory leaks),
- nerobí zlé uvoľnenie pamäte (dvojité uvoľnenie),
- neodovzdáva prekrývajúce sa bloky zdrojovej a cieľovej oblasti pamäte do funkcie `memcpy` a podobných funkcií.

Dynamické kontroly môže do programu vkladať aj priamo samotný vývojár. Väčšinou ide o príkazy typu `assert`, ktoré kontrolujú splnenie invariantov, pre-conditions alebo post-conditions. Ukážeme si to na nasledovnom programe v jazyku Python:

```
1  from math import sqrt
2
3  def fun(x):
4          assert x > 0, "x by malo byť kladné"
5          return sqrt(x)
6
7  print(fun(16))
8    # 4.0
9
10 print(fun(-1))
11   # AssertionError: x by malo byť kladné
```

Funckia `fun` očakáva kladné číslo x (lebo inak nebude vedieť vypočítať odmocninu na riadku 5). Namiesto toho, aby programátor slepo veril tomu, že pri každom zavolaní funkcie `fun` bude $x > 0$, tak na riadku 4 kontroluje, či je to naozaj tak. Ak nie, tak sa chod programu hneď zastaví a vypíše sa správa „x by malo byť kladné“. Pokiaľ sa jazyk kompiluje, tak pri

kompilácii v tzv. „Release“ móde sa všetky volania `assert` odignorujú (aby nespomaľovali chod odladenej aplikácie pri ostrom nasadení).

2.4 Znalosť programovacieho jazyka

Ako sme už spomenuli v podčasti 2.3.1, správny programátorský štýl je dôležitý pre rýchle a korektné pochopenie programu. Pod správnym štýlom ale nemusíme myslieť len predpísané formátovanie, ale aj vyhnutie sa používaniu nejednoznačných konštrukcií.

Čo napríklad autor skutočne zamýšľal v nasledovnom PHP kóde: `if (!$a)` ...? Chcel zistiť, či premenná `$a` je rovná `false`, `0` alebo `NULL`? PHP všetky tieto konštanty totiž považuje za nepravdivú hodnotu.

Iný príkladom môže byť test: `if ($a == "{}")` ... . Okrem `$a === "{}"` prejde testom aj `$a === NULL`. Počítal s týmto aj pôvodný autor? Alebo o tejto „špecialite“ použitého jazyka nevedel a môže byť potenciálnym počiatkom nejakej zraniteľnosti?

Explicitné riešenia sú vo všeobecnosti lepšie. Nevyžadujú znalosti celého kódu a rôznych špecialít použitého jazyka, čím sa predchádza zbytočným chybám.

2.5 Znalosť operačného systému

Okrem dobrej znalosti použitého programovacieho jazyka je dôležitá aj znalosť operačného systému, na ktorom bude výsledná aplikácia prevádzkovaná. Napríklad na operačnom systéme Windows sa po zadaní príkazu bez prípony postupne skúšajú prípony z premennej prostredia `PATHEXT`. Jej preddefinovaná hodnota je: `.COM; .EXE; .BAT; .CMD; .VBS; .VBE; .JS; .JSE; .WSF; .WSH; .MSC; .CPL`. Zmenou `PATHEXT` môže útočník spôsobiť spustenie inej (než očakávanej) aplikácie.

Dajme tomu, že na systéme existuje program s názvom `prikaz.exe`. Útočník nemôže tento program modifikovať, ale môže vytvoriť nový program s názvom `prikaz.com`. Keď používateľ zadá príkaz `prikaz`, tak sa operačný systém Windows pokúsi najprv spustiť program `prikaz.com` a až potom program `prikaz.exe`. Týmto spôsobom môže útočník oklamať používateľa a primäť ho k tomu, aby spustil ním zvolený program.

Podobné problémy spôsobuje aj premenná prostredia `PATH`, ktorá zas určuje poradie adresárov, v ktorých systém hľadá zadaný program. Táto premenná prostredia je relevantná aj pre UNIXové operačné systémy, napríklad Linux.

2.6 Allow vs. deny list

Bezpečnosť by sa *nemala* zakladať na vymenovaní nebezpečných vecí (black/deny list), pretože sa ľahko na niečo zabudne. Naopak, mali by sme vymenovať bezpečné veci a ostatné zakázať (white/allow list). V takomto prípade je jasne dané, čo môže programátor na vstupe

očakávať a s tým teda bude počítať. Pritom v prvom prípade je síce jasne dané, čo na vstupe určite nebude, ale nie je evidentné, čo všetko na vstupe môže byť. Ak niečo z toho môže spôsobiť neočakávané správanie, tak to môže viesť k zraniteľnosti aplikácie.

Ako príklad nesprávneho prístupu môžeme uviesť situáciu, keď sa pomocou deny listu programátor snaží zakázať vloženie skriptu do HTML kódu. Preto bude tag `<script>` a `</script>` zakázaný napríklad tým, že zo vstupného reťazca bude odstránený nahradením za prázdny reťazec. Avšak útočník môže vo vstupe zadať napríklad `<scr<script>ipt>` z ktorého po nahradení `<script>` za prázdny reťazec (`<scr` $\underbrace{\texttt{<script>}}_{\varepsilon}$ `ipt>`) ostane `<script>`.

2.7 Zraniteľnosti vo formátovacom reťazci

Mnohé jazyky priamo vo svojej špecifikácii (alebo v štandardnej knižnici) obsahujú príkaz pre výpis na konzolu. Tento výpis môže byť často naformátovaný do požadovaného tvaru. Napríklad pri výpise reálnych čísiel môže programátor požadovať výpis na tri desatinné miesta. Alebo môže uviesť celý okolitý text a miesta kam sa majú požadované údaje vložiť, ako v nasledujúcom príklade (pre jazyk C):

```
printf("Name: %s (age: %d)", person, age);
```

Name: Einstein (age: 142)

Formátovací reťazec je označený žltou farbou. Funkcia `printf` má premenlivý počet argumentov. Ak označíme p počet znakov `%` vo formátovacom reťazci, tak počet argumentov (vrátane formátovacieho reťazca) by mal byť $p + 1$. Ak je počet argumentov menší ako $p + 1$ tak sa bude funkcia `printf` snažiť pri vypisovaní pristupovať k neexistujúcim argumentom, čo môže viesť k zrúteniu aplikácie.

Z takýchto a podobných chýb vznikajú bezpečnostné zraniteľnosti, najmä ak je do časti formátovacieho reťazca vložený nedôveryhodný reťazec, ktorý napríklad pochádza priamo od používateľa. V nasledovných podčastiach si ukážeme niekoľko príkladov takýchto zraniteľností v rôznych programovacích jazykoch.

2.7.1 C

Funkcia `printf` nie je jediná, ktorá používa formátovací reťazec. Aj funkcie z rôznych knižníc môžu niektorý zo svojich argumentov (obvykle prvý) interpretovať ako formátovací reťazec. V nasledovnom príklade programátor správne a bezpečne použije funkciu `snprintf`:

```
snprintf(str, sizeof(str), "Wrong password (user %s)", user);
```

Tým si pripraví reťazec `str`, ktorého obsah chce zapísať do logov. Preto následne zavolá systémovú funkciu `syslog`:

```
syslog(LOG_WARNING, str);
```

Avšak funkcia `void syslog(int priority, const char *format, ...)` používa svoj druhý argument ako formátovací reťazec. Preto pri nasledovnom vstupe:

```
user = "einstein%s%s%s%s"
```

aplikácia veľmi pravdepodobne spadne – „Segmentation fault (core dumped)“. Šikovný kompilátor dokonca počas statickej kontroly pri kompilácii zahlási nasledovné varovanie: „format not a string literal and no format arguments [-Wformat-security]“.

Rovnakej chyby sa dopustil aj programátor, ktorý sa nasledovným programom:

```
fprintf(log, logmessage);
```

snažil vypísať reťazec `logmessage` na ďalšie miesto. V tomto prípade do súboru `log`. Správne mal napísať:

```
fprintf(log, "%s", logmessage);
```

Takýmto triviálnym formátovacím reťazcom (označený žltou farbou) sa jasne povie, že sa očakáva iba jeden ďalší parameter a tým je reťazec, ktorý sa má vypísať bez ďalšieho interpretovania jeho obsahu.

Obvzlášť nebezpečný formátovací parameter je parameter `%n`. Jeho význam je podľa špecifikácie nasledovný:

“Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored.”

To znamená, že sa nič nevypíše, ale do prislúchajúceho argumentu sa vloží číslo predstavujúce počet doteraz vypísaných znakov. Preto sa príslušný argument interpretuje ako ukazovateľ na celé číslo so znamienkom. Pôvodným dôvodom pre zavedenie takéhoto parametra bola pravdepodobne snaha uľahčiť formátovanie tabuliek. Dnes sa však tento parameter takmer vôbec nepoužíva a preto veľa programátorov o ňom nevie.

Zneužitie parametra `%n` si už ale (pre zmenu) ukážeme na programovacom jazyku Perl.

2.7.2 Perl

Majme súbor `format2.pl` s nasledovným obsahom:

```
1 #!/usr/bin/perl
2 $a = "10";
3 printf("Before: $a\n");
4 printf("$ARGV[0]", $a); # <- !!!
5 printf("After: $a\n");
```

Keď z príkazového riadku zavoláme príkaz `./format2.pl`, tak vypíše:

```
Before: 10
After: 10
```

Ak však šikovný útočník zadá príkazu vhodný parameter (`./format2.pl 123%n`), tak dokáže zmeniť hodnotu premennej `$a` a program potom vypíše:

```
Before: 10
123After: 3
```

2.7.3 PHP

Programovací jazyk PHP už nepodporuje formátovací príkaz `"%n"`. Ukážme si však zaujímavé a dobre mienené správanie jazyka PHP na nasledovnom programe (ktorý uložíme do súboru s názvom `format3.php`):

```
1 #!/usr/bin/php
2 <?php
3   printf("%s","Hello 1!\n");
4   printf("%s%s","Hello 2!\n"); # <- !!!
5   printf("%s","Hello 3!\n");
6 ?>
```

Po zadaní príkazu `./format3.php` sa vypíše:

```
Hello 1!
PHP Warning: printf(): Too few arguments in format3.php on line 4
Hello 3!
```

Autori jazyka PHP kontrolujú počet argumentov a pokiaľ je ich málo, tak vypíšu varovanie. Samotný príkaz `printf` v takomto prípade nič neurobí a program v PHP pokračuje ďalej. Týmto sa síce predíde pádu aplikácie, ale zároveň sa toto správanie dá útočníkom zneužiť napríklad na potlačenie záznamu do logov.

2.7.4 Python

Programovací jazyk Python tiež nepodporuje `"%n"`, dokonca ani `printf`. Obsahuje však podobný príkaz % (je to vlastne operátor). Nech skript `format4.py` je nasledovný:

```
1 #!/usr/bin/python
2 userdata = {"user": "admin", "pass": "usr123"}
3 passwd = raw_input("Password: ")
4 if passwd != userdata["pass"]:
5   print ("Wrong password: " + passwd) % userdata
6 else:
7   print "Welcome %(user)s!" % userdata
```

Po zadaní príkazu `./format4.py`, môže vyzerať dialóg nasledovne:

```
Password: %(pass)s
Wrong password: usr123
```

V tomto prípade útočník zadal ako heslo špeciálny reťazec „%(pass)s“. To na riadku 5 spôsobilo zmenu významu formátovacieho reťazca a v konečnom dôsledku vypísanie hodnoty asociovanej s kľúčom `pass` v slovníku `userdata` ako reťazca (`s`).

Podobne ako pri PHP aj v Pythone nesúlad počtu % a argumentov nespôsobí pád aplikácie. V tomto prípade vedie k výnimke. Toto správanie však možno tiež zneužiť na potlačenie záznamu do logov (ak je výnimka nevhodne ošetrená) alebo na DoS útok (ak je neošetrená).

2.8 Vyčerpanie zdrojov

Počítače a operačné systémy, na ktorých sú aplikácie prevádzkované, majú k dispozícii rôzne zdroje, ktoré potrebujú k ich prevádzke. Nech sú tieto zdroje akékoľvek, tak sú v praktických aplikáciách vždy v konečnom dôsledku obmedzené. Máme k dispozícii iba konečne veľkú pamäť, konečný počet procesorov, konečnú rýchlosť pripojenia a tak ďalej. Tieto konečné zdroje sú zdieľané všetkými aplikáciami, ktoré sú spustené na jednom systéme. Pokiaľ jedna aplikácia vyčerpá všetky zdroje (stačí jedného typu), tak všetky ostatné aplikácie, ktoré potrebujú ďalší takýto zdroj pre pokračovanie vo svojej činnosti, zaspia alebo havarujú. Uveďme si niektoré zdieľané zdroje, ktoré môžu byť vystavené takémuto útoku:

- operačná pamäť
  útočník alokuje veľké množstvo pamäte a neostane pre chod ostatných aplikácií
- diskový priestor
  po zaplnení disku nemôžu aplikácie vytvárať dočasné súbory, ukladať dokumenty
- prenosová kapacita
  zahltením prenosovej kapacity sa znemožní rozumná komunikácia ostatným procesom v systéme s okolím (bude dochádzať k timeout-om, strate nadviazaných spojení, ...)
- CPU
  vyťažením CPU sa výrazne spomalia všetky ostatné aplikácie
- entropia (pre generovanie náhodných čísiel)
  Niektoré počítače majú hardvérové generátory náhodných čísiel. V takomto prípade nie je problém poskytnúť takmer nevyčerpateľný tok náhodných dát. Pokiaľ však nedisponujú takýmto výdatným hardvérovým zdrojom náhodnosti, tak sa operačný systém snaží aspoň zbierať rôzne nedeterministické udalosti a z nich extrahovať entropiu, ktorú použije na generovanie náhodných dát. Keď sa entropia minie a sú požadované ďalšie náhodné dáta, môže systém blokovať aplikáciu, kým sa nenazbiera dosť entropie, alebo pokračovať ďalej, ale už len so pseudonáhodnými dátami.
- tabuľka procesov
  Operačný systém má obvykle limitovaný počet procesov, ktoré vie súčasne spustiť. Pokiaľ sa tento limit dosiahne, nie je možné spustiť ďalšie aplikácie. Aj existujúce aplikácie môžu mať problém pokračovať vo svojej činnosti, pokiaľ na to potrebujú vytvárať nové procesy (pre nové podúlohy).
- popisovače súborov
  Rovnako môže byť limitovaný počet súborov, ktoré môžu byť súčasne otvorené. Po minutí popisovačov už nie je možné otvoriť ďalšie súbory, čo vedie k pádom aplikácií.

- databázové a iné servery
  Aplikácia môže využívať rôzne ďalšie systémy pre svoju činnosť. Ich výkon, počet pripojení a podobne je tiež limitovaný. Ich vyčerpanie bude mať negatívny vplyv aj na samotnú aplikáciu.

- analytici
  V prípade útoku na systém je potrebné mať dostatočný počet analytikov, ktorí budú schopní na útok adekvátne zareagovať, systém ochrániť alebo zamedziť vzniku ďalších škôd, prípadne identifikovať útočníkov. Tí sa môžu snažiť vyťažiť analytikov viacerými inými falošnými útokmi, aby blokovali tieto ochranné opatrenia.

Dôvody pre existenciu zraniteľností vedúcich k vyčerpaniu zdrojov môžu byť rôzne. Môže ísť napríklad o chyby v implementácii, ktoré sú útočníkom premenené na zraniteľnosti. Typickým príkladom sú „memory leaks“. Ak ich útočník vie intenzívne zneužiť môžu viesť k rýchlemu vyčerpaniu operačnej pamäte. Rovnako môže ísť aj o chyby v návrhu. Napríklad absencia kontroly prístupu umožní útočníkom vyťažovať server bez toho, aby sa museli najprv autentifikovať. Dobrým zvykom by malo byť obmedzenie čerpania zdrojov (napríklad pomocou `ulimit`), aby každá aplikácia mala svoje vlastné limity čerpania zdrojov, ktoré ani v tom najhoršom prípade nemôže prekročiť a tak odstaviť celý systém.

Iným príkladom „chyby“ v návrhu sú protokoly s vnútorným stavom. Tieto sú z bezpečnostného hľadiska nutne náchylnejšie na DoS útoky, keďže s každým nadviazaným spojením si musí server pamätať aj jeho stav, ktorý ukladá napríklad do databázy. Takéto riešenie vyžaduje od servera oveľa viac zdrojov, ako keby si stav pamätať nemusel. Každý stavový protokol sa dá zmeniť na bezstavový. Myšlienka riešenia spočíva v zašifrovaní stavu a jeho odoslaní klientovi spolu s odpoveďou na jeho požiadavku. Klient bude následne povinný pri ďalšej požiadavke zašifrovaný stav poslať naspäť na server. Takéto riešenie síce nemíňa pamäť na serveri, ale zas viac zaťažuje CPU (šifrovanie) a šírku pásma (preposielanie stavu).

2.8.1 Neobozretné protokoly alebo algoritmy

Iným samostatným zdrojom problémov s vyčerpaním prostriedkov sú „neobozretné“ protokoly alebo algoritmy. Pod takúto neobozretnosť môžeme zaradiť napríklad nerozumné poradie vykonávania krokov v protokoloch. Napríklad v pôvodnej verzii protokolu pre vzdialený prístup k pracovnej ploche operačného systému Windows musel systém vykonať sériu komplexných operácií[4] ešte pred samotnou autentifikáciou používateľa. Takáto neskorá autentifikácia umožnila útočníkom značne zaťažovať systémové zdroje aj bez toho, aby mali na danom systéme vôbec vytvorené konto.

Neobozretné protokoly môžu viesť aj k znásobovaniu sily útočníka. Napríklad cez broadcast, subscriptions a tak podobne. Ide o tzv. asymetrické útoky, kde cena pre útočníka je oveľa nižšia ako pre napadnutý počítač (ochrancu). Ako príklad takéhoto útoku môžeme uviesť „smurf attack“ (šmolko-útok). Tým, že útočník spraví „ICMP ping“ na broadcast adresu s falošnou adresou odosielateľa, príde k znásobeniu jeho sily. On iba odošle jeden malý ICMP

[4] vytvorenie sedenia a grafického používateľského rozhrania

paket. Sieť ho ale rozdistribuuje na všetky lokálne počítače. Každý z nich potom samostatne odpovie na falošnú adresu, ktorá je adresou obete a tak môže prísť až k niekoľko-rádovému znásobeniu jedného paketu.

2.8.2 Útoky na algoritmickú zložitosť

K vyčerpaniu zdrojov môže ľahko prísť aj v situácii, keď sa prejaví zlá časová alebo priestorová algoritmická zložitosť použitých algoritmov. Častokrát sa používajú algoritmy, ktorých zložitosť v najhoršom prípade je horšia ako v priemernom prípade. Takého algoritmy môžu byť dostatočne efektívne pri bežnom používaní (keď sa prejavuje zložitosť priemerného prípadu). Existujú však vstupy, na ktorých sa prejaví zložitosť v najhoršom prípade. Pokiaľ útočník dokáže nájsť takéto zlé vstupy a cielene ich posielať systému, môže prísť k veľkému nárastu záťaže. Ukážme si niekoľko algoritmov, ktorých implementácie môžu mať rozdielnu časovú zložitosť v priemernom a najhoršom prípade:

Algoritmus	**Priemerný prípad**	**Najhorší prípad**
quicksort	$O(n \log n)$	$O(n^2)$
hash tabuľky	$O(n)$	$O(n^2)$
regulárne výrazy	$O(n)$	$O(2^n)$

Riešením v takomto prípade je použitie algoritmov, ktoré nie sú zraniteľné. Buďto majú v najhoršom prípade rovnako dobrú zložitosť ako v priemernom, alebo je pre ne ťažké nájsť zlé vstupy, na ktorých sa zložitosť najhoršieho prípadu prejaví[5].

Veľmi nebezpečné môžu byť zložité algoritmy použité pri implementácii nejakej neprerušiteľnej úlohy. Niečo také sa môže stať najskôr pri implementácii jadra operačného systému. Pokiaľ sú prerušenia zakázané a úloha, ktorá sa obyčajne vykonáva krátko, zrazu trvá dlhý čas, tak vlastne ochromí beh celého systému a všetkých na ňom prevádzkovaných aplikácií.

2.8.2.1 ReDoS

Typickým príkladom útoku na algoritmickú zložitosť je ReDoS (regular expression denial of service attack). Treba si uvedomiť, že existujú rôzne implementácie vyhľadávania regulárnych výrazov. Typicky začínajú preložením hľadaného regulárneho výrazu do nedeterministického konečného automatu (NKA) s m stavmi. Potom môže nasledovať:

- konverzia NKA na deterministický konečný automat (DKA)
  Konverzia môže mať v najhoršom prípade zložitosť až $O(2^m)$. Obvykle má ale zložitosť iba $O(m)$. Vyhľadávanie má potom zložitosť v najhoršom prípade iba $O(n)$, kde n je dĺžka vstupu, v ktorom sa regulárny výraz hľadá.
- backtracking cesty v NKA
  Vyhľadávanie môže mať v najhoršom prípade zložitosť až $O(2^n)$. Obvykle je však zložitosť iba $O(n)$.

[5] Napríklad univerzálne hešovanie bolo navrhnuté za týmto účelom.

- prehľadávanie všetkých ciest v NKA súčasne: $O(m^2 n)$
- „lenivá“ konverzia na DKA počas hľadania cesty: $O(m^2 n)$

Mnoho aplikácii používa napríklad backtracking cesty v NKA. Vidíme, že zložitosť je obvykle výborná – lineárna, ale v najhoršom prípade môže prísť k jej katastrofickému nárastu na exponenciálnu. Typický nebezpečný regulárny výraz je v tvare: `"^(a+)+$"`. Niekedy nemusí byť ľahké identifikovať tento vzor v zložitejšom regulárnom výraze. Napríklad v „OWASP Validation Regex Repository“ sa nachádza výraz pre „Java Classname“ v tvare `"^(([a-z])+.)+[A-Z]([a-z])+$"`, ktorý tiež obsahuje tento vzor.

Útočník môže ReDoS aplikovať, keď môže nebezpečnému regulárnemu výrazu zadať nevhodný vstup (napr. `"aaaaaaaaaa!"`) alebo vložiť nebezpečný podvýraz do väčšieho regulárneho výrazu (a potom zadať nevhodný vstup).

Uložme si nasledujúci program v jazyku Python do súboru `redos.py`:

```
1  #!/usr/bin/python
2
3  import re
4
5  def main():
6    cregex = re.compile(r"^(a+)+$")
7    cregex = re.compile(r"^(([a-z])+.)+[A-Z]([a-z])+$")
8    print("Prvé porovnanie")
9    match = cregex.match("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!")
10   print("Druhé porovnanie")
11   match = cregex.match("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!")
12   print("Koniec")
13
14 if __name__ == "__main__":
15   main()
```

Po jeho spustení príkazom `./redos.py` uvidíme exponenciálny nárast zložitosti medzi prvým a druhým porovnávaním.

2.9 Chybná správa pamäte

Chybná správa pamäte je častou príčinou pádov aplikácií. Medzi typické chyby patria:

- memory leaks
  Program na niektorých miestach zabúda uvoľňovať alokovanú pamäť. Pamäť sa tak postupne míňa, až sa zaplní a aplikácia spadne. Dnes, keď je bežne k dispozícii veľké množstvo operačnej pamäte, nemusí byť tento problém dostatočne viditeľný a môže prejsť až do produkčného nasadenia bez povšimnutia. Následne pri dlhodobej prevádzke bez reštartovania servera príde postupne k vyčerpaniu pamäte.

- viacnásobné uvoľnenie tej istej pamäte
  Opačným problémom je opakované uvoľnenie už uvoľnenej pamäte, ktoré podľa špecifikácie vedie k nedefinovanému správaniu (môže napríklad prísť k poškodeniu dátových štruktúr správy pamäte).

- použitie už uvoľnenej pamäte
  Pokiaľ bude program naďalej pracovať so smerníkmi, ktoré ukazujú do predtým už uvoľnenej pamäte, tak môže prísť k poškodeniu nových dát, ktoré mohli byť v tomto priestore alokované. Rovnako čítanie z takejto oblasti môže viesť k prečítaniu nezmyselných informácií.

- uvoľnenie nesprávnej pamäte
  Táto situácia spôsobuje podobné problémy ako predošlý prípad.

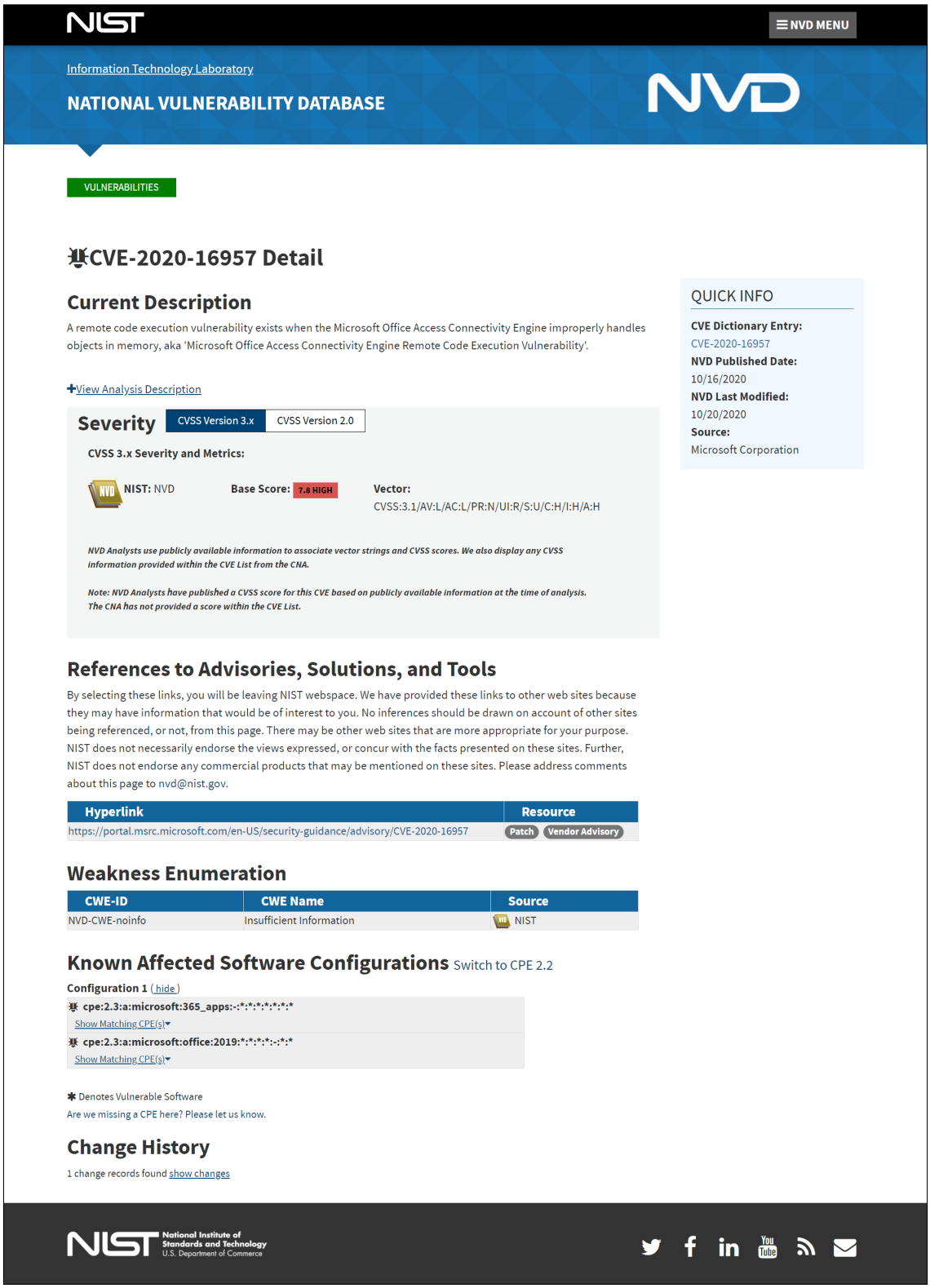
- prístup k pamäti na neplatnej adrese
  Pokiaľ sa použije smerník, ktorý je výsledkom nejakého výpočtu, tak nemusí byť garantované, že ukazuje na platnú adresu. Pokiaľ tomu tak nie je, tak dereferencovanie takéhoto smerníka povedie k porušeniu ochrany pamäte a ukončeniu aplikácie.

Pri nesprávnom návrhu aplikácie a nakonfigurovaní operačného systému môže prísť k úniku dôverných informácií uložených v pamäti programu. Treba dať pozor na odloženie obsahu operačnej pamäte s citlivou informáciou[6] na disk. Jedným z dôvodov pre odloženie časti obsahu operačnej pamäte na disk môže byť virtuálna pamäť, keď časť operačnej pamäte sa môže dočasne odložiť do odkladacieho priestoru / stránkovacieho súboru („swap space“ / „page file“). V tomto prípade sa dá úspešne použiť zamykanie stránok v operačnej pamäti, čím sa práve predíde ich odloženiu na disk.

Iným prípadom, kedy sa obsah pamäte môže dostať na disk, je pád aplikácie. Vtedy operačný systém môže na disk odložiť obsah pamäte aplikácie v okamihu jej pádu pre lepšie odladenie chyby, ktorá pád spôsobila. Takéto súbory sa na nazývajú „crash dumps“ alebo „core files“. V nastavení operačného systému sa dá zakázať vytváranie týchto súborov. Aj keď sú dôležité počas vývoja aplikácie, v produkčnom prostredí môžu byť práve z vyššie popísaných dôvodov nebezpečné.

Dobrou praxou je uchovávať citlivú informáciu v pamäti čo najkratší čas. Akonáhle už nie je potrebná, treba ju v pamäti prepísať, aby sa minimalizovala pravdepodobnosť jej úniku, nech už by to bolo akýmkoľvek spôsobom.

[6] napríklad heslá alebo šifrovacie kľúče



Obr. 2.4: Záznam k zraniteľnosti „CVE-2020-16957“ v NVD

Kapitola 3

Záver

Naším cieľom bolo poukázať na dôležitosť bezpečného programovania. Ako vidíme z verejných databáz zraniteľností, ani veľké softvérové firmy sa nevyhnú chybám vo svojich masovo rozšírených produktoch.

Nechceli sme však ostať len pri konštatovaní dôležitosti. Preto sme ukázali aj niekoľko prístupov k zvýšeniu bezpečnosti výsledného kódu, ako napríklad návrh, vývoj a nasadenie aplikácie s ohľadom na bezpečnosť počas celého životného cyklu softvéru. Príkladom môže byť statická a dynamická analýza kódu počas jeho vývoja. Dôležitá je aj znalosť rôznych drobných, ale z bezpečnostného hľadiska dôležitých vlastností použitého programovacieho jazyka alebo operačného systému.

Bezpečnosť výsledného riešenia je zásadne ovplyvnená už počas návrhu aplikácie. Je prirodzené, že sa pri ňom zohľadňujú očakávané spôsoby jej použitia. Pre bezpečnosť je však rovnako dôležité zvážiť aj neočakávané možnosti jej zneužitia (napríklad pre zosilnenie útokov, vyčerpanie zdrojov a podobne).

Bibliografia

[1] D. Deogun, D. B. Johnsson a D. Sawano. *Secure by design*. Shelter Island: Manning Publications, 2019. ISBN: 9781617294358 (citované na strane 2).

[2] M. Howard a D. LeBlanc. *Writing secure code*. 2. vydanie. Redmond, Wash: Microsoft Press, 2003. ISBN: 9780735617223 (citované na strane 2).

[3] B. Chess a J. West. *Secure programming with static analysis*. Addison-Wesley software security series. Upper Saddle River, NJ: Addison-Wesley, 2007. ISBN: 9780321424778 (citované na strane 2).

[4] F. Long. *Java coding guidelines: 75 recommendations for reliable and secure programs*. The SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN: 9780321933157 (citované na strane 2).

[5] M. S. Merkow a L. Raghavan. *Secure and resilient software development*. Boca Raton, FL: CRC Press, 2010. ISBN: 9781439826966 (citované na strane 2).

[6] *Secure Coding Guidelines for Java SE*. Technická správa. Oracle, sept. 2020. URL: https://www.oracle.com/java/technologies/javase/seccodeguide.html (citované na strane 2).

[7] *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*. Technická správa. Pittsburgh, PA: Carnegie Mellon University – Software Engineering Institute, jún 2016. URL: https://resources.sei.cmu.edu/forms/secure-coding-form.cfm (citované na strane 2).

[8] *SEI CERT Coding Standards*. Technická správa. Pittsburgh, PA: Carnegie Mellon University – Software Engineering Institute, 2020. URL: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards (citované na strane 2).

[9] J. Viega a M. Messier. *Secure programming cookbook for C and C++*. 1. vydanie. Cambridge: O'Reilly, 2003. ISBN: 9780596003944 (citované na strane 2).

[10] D. A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Mar. 2003. URL: https://tldp.org/HOWTO/pdf/Secure-Programs-HOWTO.pdf (citované na strane 2).