EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

jihomoravský kraj

BEZPEČNÉ PROGRAMOVÁNÍ

Softwarové bezpečnostní chyby

Metodický list

Autor: doc. Ing. Jaroslav Dočkal, CSc., Metodik: Ing. Miluše Jašková

Recenzent: Ing. Petr Skyva

Rok vydání: 2023

Obsah

Dovednosti 3

Pracovní prostředí 3

Průběh výuky 4

1 SOFTWAROVÉ BEZPEČNOSTNÍ CHYBY 4

1.1 Chyba zabezpečení: Problémy s vyrovnávací pamětí 5

1.2 Chyba zabezpečení: Řízení přístupu 5

1.3 Chyba zabezpečení: Vystavení informacím 5

1.4 Chyba zabezpečení: Nesprávné ověření vstupu 6

1.5 Chyba zabezpečení: Nesprávné řízení přístupu 6

1.6 Chyba zabezpečení: Chyby správy zdrojů 7

1.7 Chyba zabezpečení: Kód 7

1.8 Chyba zabezpečení: Vložení kódu 8

1.9 Chyba zabezpečení: Kryptografické problémy 8

1.10 Chyba zabezpečení: Numerická chyba 8

2 HLEDÁNÍ CHYB V PROGRAMECH 10

2.1 Problém vyrovnávací paměti: gnu libc 10

2.1.1 Chyba v programu 10

2.1.2 Oprava 10

2.2 Problém řízení přístupu v rámci Kerbera 11

2.2.1 Chyba v programu 11

2.2.2 Oprava 12

2.3 Příklad chybného vystavení informací: ADOBE AIR 12

2.3.1 Chyba v programu 12

2.3.2 Oprava 13

2.4 Příklad nesprávného ověření vstupních dat: SQLite 13

2.4.1 Chyba v programu ... 13
2.4.2 Oprava ... 14
2.5 Příklad nesprávného řízení přístupu: stunnel ... 15
2.5.1 Chyba v programu ... 15
2.5.2 Oprava ... 16
2.6 Příklad chybné správy zdrojů: Privoxy ... 17
2.6.1 Chyba v programu ... 17
2.6.2 Oprava ... 17
2.7 Příklad zranitelnosti kódu: CHRONY (NTP) ... 18
2.7.1 Chyby v programu ... 18
2.7.2 Oprava ... 19
2.8 Příklad zranitelnosti kódu: Injekce kódu ... 20
2.8.1 Chyby v programu ... 20
2.8.2 Oprava ... 20
2.9 Příklad kryptografických problémů: Použití OPENSSL ... 20
2.9.1 Chyby v programu ... 20
2.9.2 Oprava: vždy volat funkci náhodně ... 21
2.10 Příklad numerické chyby: QT ... 22
2.10.1 Chyby v programu ... 22
2.10.2 Oprava ... 23
Shrnutí a závěr ... 24
Seznam použitých zdrojů ... 27

Cíle

- Seznámit se s nejčastějšími bezpečnostními chybami v oblasti programování vložených systémů.
- Porozumění příčinám typických chyb ve vložených (embedded) systémech.
- Kriticky se vyjadřovat k příčinám programátorských chyb.
- Diskutovat, kam až sahá kompetence programátora a od kterého momentu je vhodná spolupráce s bezpečnostním specialistou.

Dovednosti

Identifikace bezpečnostních problémů v jazycích C/C++ v akutních prostředích

- SQLi
- Kerberos
- Adobe Flash Player
- stunnel
- chrony
- wrapery
- Privoxy
- OpenSSL
- dekodér Qt

Pracovní prostředí

Úlohu lze realizovat v prostředí:

- Cylab JCEKB
- Offline Security Classroom

Průběh výuky

1 SOFTWAROVÉ BEZPEČNOSTNÍ CHYBY

Jako motto modulu zde bude citováno z (Buchanan[1], 2022):

„C a C++ v podstatě pocházejí z jednodušší doby. Bylo to kódování pro inženýry, aby si s kódem dělali, co chtěli. A tak nyní Mark Russinovich – technologický ředitel Microsoft Azure – řekl, že vývojáři by měli přestat používat C a C++ a zaměřit se na Rust. Proč, no, hlavně kvůli problémům s bezpečností a spolehlivostí. C a C++ se opravdu nedá věřit.“

……

„Nedávno bylo zjištěno, že přibližně 70 % všech chyb paměti lze vysledovat do C a C++ a přibližně stejnou úroveň chyb v Chrome lze vysledovat do C a C++. Mark Russinovich ve skutečnosti říká, že C a C++ by měly být jako zastaralé odstraněny z oblasti budování softwarových systémů. Ale stále je to jeden z nejvíce vyučovaných programovacích jazyků a stále je to jeden z nejžádanějších jazyků, takže je možná potřeba něco změnit na základní úrovni, aby se softwarový průmysl posunul kupředu.

Spolu s Microsoftem, který přesouvá svůj kód do Rust, Meta a AWS také přesouvají back-endový kód (na straně serveru) na Rust, zatímco jiné společnosti se snaží přidat do kódu C++ možnosti bezpečnostní kontroly pro paměť. Celkově se však do C++ investuje příliš mnoho na to, abychom se něj v současné době vzdali, a pravděpodobně potrvá desetiletí, než uvidíme poslední C++.“

Obsahem modulu je deset nejčastějších bezpečnostních chyb v oblasti programování vložených (embedded) systémů. Přesná definice pro určení toho, co je vnořený systém, neexistuje. Náplň pojmu nelze přesně určit, ale mezi jeho hlavní rysy patří to, že jde o kombinaci hardwaru a softwaru provázanou jako celek, určenou většinou pro konkrétní úkoly.

Při zpracování metodického listu se vycházelo z materiálů společnosti Perforce Software (Ashley, 2021), (Perforce 2011a), (Perforce 2011a) a (Perforce 2011a). Z webu této firmy si lze stáhnout potřebný nástroj pro vyzkoušení statické analýzy programu[2] (bezplatně na týden).

[1] Profesor kryptografie na University of Edinburgh.
[2] https://www.perforce.com/products/klocwork/demo?utm_term=trendemon

1.1 Chyba zabezpečení: Problémy s vyrovnávací pamětí

K problémům s vyrovnávací pamětí dochází, když software může číst nebo zapisovat do umístění mimo hranice vyrovnávací paměti.

To se může stát v těchto případech:

- Nekontroluje se velikost vstupu na kopii.
- Chyba umožňující zápis do libovolných umístění.
- Čtení mimo hranice.
- Ukazatele mimo očekávaný rozsah.
- Nedůvěryhodné dereference[3] ukazatele.
- Neinicializované ukazatele.
- Prošlé odkazy na ukazatele.
- Přístup k paměti za koncem vyrovnávací paměti.

1.2 Chyba zabezpečení: Řízení přístupu

Řízení přístupu je jakákoli slabina související se správou oprávnění, privilegii nebo jiných funkcemi zabezpečení. To se může stát v těchto případech:

- Problémy s izolovaným prostorem (chroot prostředí).
- Problémy s oprávněními (nesprávná dědičnost, povolená výchozí nastavení, symbolické odkazy podvracející oprávnění, nesprávné zachování oprávnění atd.).
- Nesprávná správa vlastnictví.
- Nesprávná kontrola přístupu.

1.3 Chyba zabezpečení: Vystavení informacím

Vystavení informací je úmyslné nebo neúmyslné zpřístupnění informací aktérovi, který k tomu není výslovně oprávněn. To se může stát v těchto případech:

- Odeslaná data.
- Datové dotazy.
- Nesrovnalosti.
- Chybové zprávy.
- Zprávy ladění.

[3] Zatímco operátor reference umožňuje uložit do ukazatele adresu proměnné, operátor dereference umožňuje přistoupit k hodnotě proměnné, na kterou ukazatel ukazuje. Zrada je v tom, že proměnným mohou být přiřazeny hodnoty bez použití přiřazovacích příkazů.

- Procesní prostředí.
- Ukládání do mezipaměti.
- Indexování soukromých dat.

Vývojáři mají tendenci zapomínat na vystavení informací prostřednictvím chybových zpráv a zpráv o ladění, stejně jako na jakékoli informace, které se mohou dostat do souboru protokolu. Z tohoto důvodu si vývojáři musí dávat pozor na vše, co se nachází v souboru protokolu, a být si jisti, že tam nejsou žádné informace, které by mohly být pro útočníka cenné.

1.4 Chyba zabezpečení: Nesprávné ověření vstupu

Nesprávné ověření vstupu zahrnuje získání nesprávných nebo chybějících informací z čehokoli, co by mohlo ovlivnit řídicí tok programu nebo tok dat. To se může stát v těchto případech:

- Nesprávná omezení názvu cesty.
- Nesprávné rozlišení ekvivalence názvu cesty.
- Externí kontrola nastavení konfigurace.
- Nesprávná neutralizace (vkládání příkazů, vkládání SQL, skriptování mezi stránkami atd.).
- Chybějící ověření XML.
- Nesprávná neutralizace polena.
- Nesprávné omezení na meze vyrovnávací paměti.
- Chybné ověření indexu pole.
- Kopírovat do vyrovnávací paměti bez kontroly velikosti.
- Nesprávné nulové ukončení.

Tato chyba zabezpečení se může vyskytnout v podstatě kdekoliv, kde vývojář získává informace z vnějšího světa.

1.5 Chyba zabezpečení: Nesprávné řízení přístupu

Nesprávná kontrola přístupu označuje situaci, kdy software neomezuje nebo nesprávně omezuje přístup ke zdroji od neoprávněného aktéra. To se může stát v těchto případech:

- Nesprávná správa oprávnění.
- Nesprávná správa vlastnictví.
- Nesprávné oprávnění.
- Nesprávná správa uživatelů.
- Nesprávné ověření.
- Chyba ověření původu.

- Nesprávné omezení komunikačního kanálu na zamýšlené koncové body.

Nesprávné omezení komunikačních kanálů je mnoha lidem známé, protože bylo součástí hacku Miller/Valasek[4]. V tomto konkrétním případě šlo o to, že služba D-Bus zůstala otevřená na mobilním datovém připojení. Ponechat službu D-Bus otevřenou z mobilního kanálu bylo velkým nedopatřením, které by bylo zachyceno při správném návrhu zabezpečení.

1.6 Chyba zabezpečení: Chyby správy zdrojů

Chyby správy zdrojů mohou odkazovat na řadu věcí, včetně:

- Nesprávná správa systémových prostředků.
- Nekontrolovaná spotřeba zdrojů.
- Přenos soukromých zdrojů do nové sféry.
- Nesprávné uvolnění nebo vypnutí zdrojů.
- Asymetrická spotřeba zdrojů.
- Uzamčení zdroje.
- Double-free, použití po free[5].
- Nedostatečný fond zdrojů.
- Absence haldy paměti.

I když jsou tyto problémy mnohem častější v jazycích jako C a C++, je vždy důležité je mít na paměti. Lidé kódující v Pythonu a Javě si často myslí, že jsou imunní vůči chybám správy zdrojů, ale to prostě není pravda. Je velmi snadné se dostat na místo, kde se paměť vyčerpává v těchto jazycích, aniž byste si uvědomovali, že se to děje.

1.7 Chyba zabezpečení: Kód

Může se zdát zvláštní mít kód jako svou vlastní zranitelnost, ale mluvíme o čemkoli, co nespadá do konkrétní kategorie – takže to považujte za univerzální zranitelnost. To může zahrnovat věci jako:

- Špatná správa hesel, ukládání hesel ve formátu prostého textu, pevně zakódovaných hesel.
- Nesprávné zacházení se smlouvami API.
- Nesprávné nebo chybějící zpracování chyb.
- Nesprávné zacházení s časem a stavem.
- Problémy s generováním kódu.

[4] Charlie Miller a Chris Valasek v roce 2015 na dálku hackli Jeep Cherokee. Fiat Chrysler stáhl 1,4 milionu vozidel a zaplatil 105 milionů dolarů jako pokuty National Highway Traffic and Safety Administration.
[5] Double-free, technicky vzato, vede k nedefinovanému chování.

Je důležité, aby si vývojáři nevytvářeli vlastní šifrování – nestojí to za to. Reverzní inženýrství je velmi snadné a mnoho lidí to opustilo frustrovaní a v rozpacích, když se nechová tak, jak očekávali.

1.8 Chyba zabezpečení: Vložení kódu

Vkládání kódu je něco, co ovlivňuje interpretovaná prostředí, jako je PHP, což z něj dělá velkou zranitelnost mezi komunitou vývojářů webových stránek. Navzdory této zranitelnosti, která je z velké části vidět při vývoji webových stránek, je stále velmi přítomná, pokud jde o informační a zábavní systémy a další komplikované vestavěné systémy.

Protože běžně existuje skriptovací stroj, který se postará o velké množství spouštění služby, obvykle existuje další vektor, který může být napaden. To může ovlivnit i black box komponenta obsahující interprety, o kterých vývojář nemusí vědět.

1.9 Chyba zabezpečení: Kryptografické problémy

S použitím kryptografií souvisí řada slabin, a to mate mnoho vývojářů. Existuje mnoho věcí, které by mohly zkazit správné použití kryptografie:

- Chybějící šifrování citlivých dat.
- Chyby správy klíčů.
- Chybějící požadovaný kryptografický krok.
- Nedostatečná síla šifrování.
- Použití poškozených nebo riskantních kryptografických algoritmů.
- Použití reverzibilního jednosměrného hashe.
- Nesprávné použití náhodného inicializačního vektoru.
- Nesprávné použití algoritmu RSA.

1.10 Chyba zabezpečení: Numerická chyba

Číselné chyby mohou odkazovat na několik různých kategorií problémů, včetně:

- Obtékání chyb.
- Nesprávné ověření indexu pole.
- Přetečení celého čísla.
- Nesprávné řazení bajtů.
- Nesprávný převod mezi číselnými typy.
- Nesprávný výpočet.

Běžné místo pro zranitelnost numerických chyb je v matematických výpočtech. Tomuto druhu zranitelnosti také podléhají data, pokud přitékají z externího zdroje.

2 HLEDÁNÍ CHYB V PROGRAMECH

2.1 Problém vyrovnávací paměti: gnu libc[6]

2.1.1 Chyba v programu

Obrázek 2.1.1 ukazuje příklad problém s vyrovnávací pamětí v GNU libc. Tento příklad může být překvapením, ale i vysoce testované, vysoce využívané a prověřované knihovny mohou mít bezpečnostní chyby. To ukazuje, že žádný vývojář ani program není imunní vůči chybám nebo problémům s kódem. Tento konkrétní příklad obsahuje wscanf[7] alokující příliš málo paměti za určitých podmínek. Jak je vidět na obrázku 1, je to opravdu všechno o tom, zda vývojáři zohledňují široké znaky.

```
if (__glibc_unlikely (wpsize == wpmax))                    \
 {                                                         \
   CHAR_T *old = wp;                                       \
   size_t newsize = (UCHAR_MAX + 1 > 2 * wpmax             \
                     ? UCHAR_MAX + 1 : 2 * wpmax);         \
   if (use_malloc || !__libc_use_alloca (newsize))         \
     {                                                     \
       wp = realloc (use_malloc ? wp : NULL, newsize);     \
       if (wp == NULL)                                     \
         {                                                 \
           if (use_malloc)                                 \
             free (old);                                   \
           done = EOF;                                     \
           goto errout;                                    \
         }                                                 \
       if (! use_malloc)                                   \
         MEMCPY (wp, old, wpsize);                         \
       wpmax = newsize;                                    \
       use_malloc = true;                                  \
     }                                                     \
   else                                                    \
     {                                                     \
       size_t s = wpmax * sizeof (CHAR_T);                 \
       wp = (CHAR_T *) extend_alloca (wp, s,               \
                                      newsize * sizeof (CHAR_T));  \
       wpmax = s / sizeof (CHAR_T);                        \
       if (old != NULL)                                    \
         MEMCPY (wp, old, wpsize);                         \
```

Obrázek 2.1.1: Příklad selhání GNU libc

2.1.2 Oprava

Široké znaky jsou vícejazykové kódy znaků, které jsou vždy 16 bitů široké a typu **`wchar_t`**. Oprava, jak je zobrazena na obrázku 2.1.2, je v podstatě jen vložení několika funkcí velikosti {CHAR_T} do kódu. Takže místo přidělování počtu bajtů by vývojáři měli přidělovat velikost mezinárodních znaků.

[6] libc je obecný termín používaný k označení všech standardních knihoven jazyka C. Navzdory svému názvu nyní přímo podporuje také C++ (a nepřímo i další programovací jazyky).
[7] Funkce wscanf() čte data ze stdin a ukládá hodnoty do příslušných proměnných.

```
if (__glibc_unlikely (wpsize == wpmax))                                 \
 {                                                                      \
   CHAR_T *old = wp;                                                    \
   bool fits = __glibc_likely (wpmax <= SIZE_MAX / sizeof (CHAR_T) / 2); \
   size_t wpneed = MAX (UCHAR_MAX + 1, 2 * wpmax);                      \
   size_t newsize = fits ? wpneed * sizeof (CHAR_T) : SIZE_MAX;         \
   if (!__libc_use_alloca (newsize))                                    \
     {                                                                  \
       wp = realloc (use_malloc ? wp : NULL, newsize);                  \
       if (wp == NULL)                                                  \
         {                                                              \
           if (use_malloc)                                              \
             free (old);                                                \
           done = EOF;                                                  \
           goto errout;                                                 \
         }                                                              \
       if (! use_malloc)                                                \
         MEMCPY (wp, old, wpsize);                                      \
       wpmax = wpneed;                                                  \
       use_malloc = true;                                               \
     }                                                                  \
   else                                                                 \
     {                                                                  \
       size_t s = wpmax * sizeof (CHAR_T);                              \
       wp = (CHAR_T *) extend_alloca (wp, s, newsize);                  \
       wpmax = s / sizeof (CHAR_T);                                     \
       if (old != NULL)                                                 \
         MEMCPY (wp, old, wpsize);                                      \
```

Obrázek 16: Příklad opravy GNU libc

2.2 Problém řízení přístupu v rámci Kerbera

2.2.1 Chyba v programu

Jak je vidět na obrázku 2.2.1, Kerberos správně nesleduje, zda byl požadavek klienta ověřen. Jde přitom o běžný protokol, který se používá pro klienty a servery ke vzájemné autentizaci, který může být místem, kde se vyskytují zranitelnosti.

```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
        testcase( (flags1&MEM_Dyn) != (pIn1->flags&MEM_Dyn) );
        flags1 = (pIn1->flags & ~MEM_TypeMask) | (flags1 & MEM_TypeMask);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
        testcase( (flags3&MEM_Dyn) != (pIn3->flags&MEM_Dyn) );
        flags3 = (pIn3->flags & ~MEM_TypeMask) | (flags3 & MEM_TypeMask);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */
assert( (pIn1->flags & MEM_Dyn) == (flags1 & MEM_Dyn) );
pIn1->flags = flags1;
assert( (pIn3->flags & MEM_Dyn) == (flags3 & MEM_Dyn) );
pIn3->flags = flags3;
break;
```

Obrázek 2.2.1: Příklad selhání protokolu Kerberos

2.2.2 Oprava

Obrázek 2.2.2 ukazuje, že za účelem vyřešení tohoto problému by vývojáři neměli předpokládat, že výchozí stav je předem autorizován. Pokud kód neautorizujete včas, budou chyby zachyceny dříve, než bude kód prohlášen za autorizovaný. Problém není nesnadné opravit.

```
static void
on_response(void *data, krb5_error_code retval, otp_response response)
{
    struct request_state rs = *(struct request_state *)data;

    free(data);

    if (retval == 0 && response != otp_response_success)
        retval = KRB5_PREAUTH_FAILED;

    if (retval == 0)
        rs.enc_tkt_reply->flags |= TKT_FLG_PRE_AUTH;

    rs.respond(rs.arg, retval, NULL, NULL, NULL);
}

static void
otp_verify(krb5_context context, krb5_data *req_pkt, krb5_kdc_req *request,
           krb5_enc_tkt_part *enc_tkt_reply, krb5_pa_data *pa,
           krb5_kdcpreauth_callbacks cb, krb5_kdcpreauth_rock rock,
           krb5_kdcpreauth_moddata moddata,
           krb5_kdcpreauth_verify_respond_fn respond, void *arg)
{
    krb5_keyblock *armor_key = NULL;
    krb5_pa_otp_req *req = NULL;
    struct request_state *rs;
    krb5_error_code retval;
    krb5_data d, plaintext;
    char *config;

    //DELETED: enc_tkt_reply->flags |= TKT_FLG_PRE_AUTH;
```

Obrázek 14: Příklad opravy Kerbera

Abyste měli stále na starosti zranitelnosti řízení přístupu, je důležité mít architekturu a design na prvním místě. Vždy je důležité korigovat konstrukcí a ujistit se, že kód funguje od začátku optimálně. Nezapomeňte také použít manuální analýzu kódu a pečlivě prozkoumejte veškeré udělení a předání řízení přístupu pro předpoklady, výchozí hodnoty a cesty chyb.

Nezapomeňte na svůj kód použít další nástroje – jako je statická a dynamická analýza. Fuzz testování a kontextově závislá analýza slabých stránek by mohly napomoci řadě problémům.

2.3 Příklad chybného vystavení informací: ADOBE AIR

2.3.1 Chyba v programu

Adobe AIR je zajímavým příkladem toho, proč byste neměli informace jen tak rozdávat. Adobe Flash Player neomezoval správně zjišťování adres paměti, což by útočníkovi umožnilo obejít ochranný mechanismus randomizace rozvržení adresního prostoru (ASLR[8]) prostřednictvím nespecifikovaných vektorů. Díky tomu je proces ukládání paměti na náhodná místa plýtváním, protože útočníci jsou schopni najít veškerou paměť.

[8] Address space layout randomization (ASLR je metoda počítačové bezpečnosti, která umisťuje strojový kód programu, knihovny a data v operační paměti od náhodně zvolené adresy. Cílem je znemožnit některé typy útoků a exploitů.

I když se to může zdát jako malý problém, je důležité si uvědomit, že většina útočníků potřebuje více než jednu zranitelnost, aby mohla zasáhnout. To je důvod, proč si vývojáři opravdu musí dávat pozor na to, jakou paměť uvolňují, aby útočníci nebyli schopni získat informace spolu s dalšími informacemi nebo jinou zranitelností.

2.3.2 Oprava

Rozdělením systémů na bezpečné oblasti, tím, že nedovolíte, aby citlivá data opustila hranice důvěry, a opatrností při propojování mimo tyto oblasti důvěry, mohou vývojáři bezpečně navrhnout kód tak, aby byl chráněn před těmito informacemi.

K zabezpečení informací by měla být také použita analýza kódu. Automatizované SCA (Strong Customer Authentication) může provádět kontextově závislou analýzu slabých stránek, dynamická analýza kódu může provádět fuzz testování, které umožňuje testování v rámci monitorovaného virtuálního prostředí. Manuální analýza kódu je stále potřebná k tomu, aby se zajistilo, že neexistují žádné oblasti vnějšího kontaktu pro nezamýšlený přístup k datům a že nebudou sdíleny žádné informace, pokud to není nezbytně nutné. Kromě toho sem patří rovněž chybové a ladicí zprávy.

2.4 Příklad nesprávného ověření vstupních dat: SQLite

2.4.1 Chyba v programu

Jeden příklad nesprávného ověřování vstupu se vyskytuje v SQLite, balíčku, který se používá v mnoha vestavěných systémech. SQLite správně neimplementuje operátory porovnání, což umožňuje kontextově závislým útočníkům způsobit odmítnutí služby nebo případně mít nespecifikované jiné dopady prostřednictvím vytvořené klauzulí CHECK. Příklad, který je vidět na obrázku 2.4.1, je trochu komplikovaný a možná trochu paranoidní, ale přesto něco, co je třeba brát vážně.

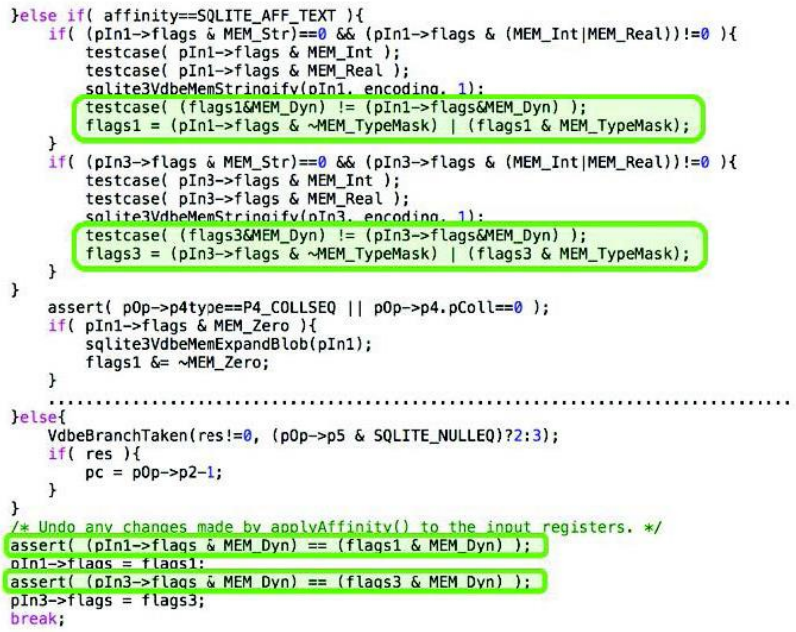
```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */

pIn1->flags = flags1;

pIn3->flags = flags3;
break;
```

Obrázek 2.4.1: Příklad selhání SQLite

Chyba na obrázku 2.4.1 spočívá v tom, že je přiřazeno několik příznaků, které končí zrušením akcí provedených rutinou. Celkovým problémem je zde špatná cesta přes kód, která může skončit resetováním těchto příznaků a vytvořením paměť vypadá, jako by byla dynamicky alokována nebyl, nebo naopak. To je samozřejmě špatné, protože útočník by mohl tuto chybu zabezpečení využít k získání informací, ke kterým by neměl mít přístup, nebo by mohl způsobit uvolnění paměti, která neměla být uvolněna.

2.4.2 Oprava

Oprava této zranitelnosti, jak je vidět na obrázku 2.4.2, je trochu komplikovaná, ale v zásadě je třeba udělat to, že oblasti, které mají příznak (flag), musí pokaždé ušetřit důležité bity pro přímý přístup do paměti. To zajistí, že když si vývojáři začnou hrát s více hodnotami příznaků, nebudou moci resetovat konkrétní bity.



```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
        testcase( (flags1&MEM_Dyn) != (pIn1->flags&MEM_Dyn) );
        flags1 = (pIn1->flags & ~MEM_TypeMask) | (flags1 & MEM_TypeMask);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
        testcase( (flags3&MEM_Dyn) != (pIn3->flags&MEM_Dyn) );
        flags3 = (pIn3->flags & ~MEM_TypeMask) | (flags3 & MEM_TypeMask);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */
assert( (pIn1->flags & MEM_Dyn) == (flags1 & MEM_Dyn) );
pIn1->flags = flags1;
assert( (pIn3->flags & MEM_Dyn) == (flags3 & MEM_Dyn) );
pIn3->flags = flags3;
break;
```

Obrázek 2.4.2: Příklad opravy SQLite

Příznak na obrázku 2.4.2 je však ve skutečnosti skutečně náchylný k chybám při používání a mohl by použít přepracování, aby bylo zajištěno, že bude fungovat bezpečně. Nesprávné chyby ověření vstupu lze často opravit správnou architekturou a návrhem, správnou implementací a využitím automatizované statické analýzy. Při navrhování kódu nezapomeňte zkontrolovat data na straně klienta i serveru transakce – nepředpokládejte, že strana serveru pouze předává zamýšlené informace, ve skutečnosti se ujistěte, že tomu tak je.

Nástroje SCA jsou velmi dobré při hledání nečistých, nekotovaných[9], nebo neescapovaná[10] data, takže se ujistěte, že používáte pro navrhování kódu nástroj SCA.

2.5 Příklad nesprávného řízení přístupu: stunnel

2.5.1 Chyba v programu

Příkladem nesprávného řízení přístupu, jak je vidět na obrázku 2.5.1, je stunnel (TLS proxy)[11]. V tomto případě se jedná o software, který lze použít k vytvoření připojení VPN, aby mohlo existovat zabezpečené připojení k back-endu. V této konkrétní chybě zabezpečení použití možnosti přesměrování ve skutečnosti nepřesměruje

[9] Zveřejněná cena na nákup a cena na prodej daného cenného papíru.
[10] Escapování je převod znaků majících v daném kontextu speciální význam na jiné odpovídající sekvence. Příklad: do řetězce ohraničeného uvozovkami chceme zapsat uvozovky. Jelikož uvozovky mají v kontextu řetězce speciální význam a jejich prosté zapsání by bylo chápáno jako ukončení řetězce, je potřeba je zapsat jinou odpovídající sekvencí. Podobně postupujeme při psaní bannerů.
[11] Stunnel je open-source multiplatformní aplikace používaná k poskytování univerzální služby tunelování TLS/SSL. Stunnel spoléhá na knihovnu OpenSSL.

připojení klientů na očekávaný server po počátečním připojení. To umožní vzdáleným útočníkům obejít ověřování.

```
if(SSL_session_reused(c->ssl)) {
    s_log(LOG_INFO, "SSL %s: previous session reused",
          c->opt->option.client ? "connected" : "accepted");
} else { /* a new session was negotiated */
    new_chain(c);
    if(c->opt->option.client) {
        s_log(LOG_INFO, "SSL connected: new session negotiated");
        enter_critical_section(CRIT_SESSION);
        old_session=c->opt->session;
        c->opt->session=SSL_get1_session(c->ssl); /* store it */
        if(old_session)
            SSL_SESSION_free(old_session); /* release the old one */
        leave_critical_section(CRIT_SESSION);
    } else
        s_log(LOG_INFO, "SSL accepted: new session negotiated");
    print_cipher(c);
}
```

Obrázek 2.5.1: Příklad selhání stunelu

2.5.2 Oprava

Jak je vidět na obrázku 2.5.1, funkce SSL volá znovu použitou relaci, ale znovu ji nekontroluje, aby se ujistila, že ji lze znovu použít. Oprava je vidět na obrázku 2.5.2 a ukazuje, proč je důležité testovat každou relaci. Pokud relace není platná, lze příkaz zachránit, aby se k zabezpečenému připojení nedostali žádní nežádoucí útočníci. Tato oprava umožňuje ověřit, že každý, kdo se přihlásí do systému, tam může být. Toto je velmi dobrý příklad toho, proč by se v kódu nikdy neměly používat zkratky.

```
s_log(LOG_INFO, "SSL %s: %s",
      c->opt->option.client ? "connected" : "accepted",
      SSL_session_reused(c->ssl) ?
      "previous session reused" : "new session negotiated");
if(SSL_session_reused(c->ssl)) {
    c->redirect=(uintptr_t)SSL_SESSION_get_ex_data(SSL_get_session(c->ssl),
                                                   redirect_index);
    if(c->opt->redirect_addr.names && !c->redirect) {
        s_log(LOG_ERR, "No application data found in the reused session");
        longjmp(c->err, 1);
    }
} else { /* a new session was negotiated */
    new_chain(c);
    SSL_SESSION_set_ex_data(SSL_get_session(c->ssl),
                            redirect_index, (void *)c->redirect);
    if(c->opt->option.client) {
        enter_critical_section(CRIT_SESSION);
        old_session=c->opt->session;
        c->opt->session=SSL_get1_session(c->ssl); /* store it */
        if(old_session)
            SSL_SESSION_free(old_session); /* release the old one */
        leave_critical_section(CRIT_SESSION);
    } else { /* SSL server */
        SSL_CTX_add_session(c->opt->ctx, SSL_get_session(c->ssl));
    }
    print_cipher(c);
}
```

Obrázek 2.5.2: Příklad opravy stunnelu

Všechny problémy s nesprávným řízením přístupu lze vyřešit pečlivou kontrolou nastavení, správou a manipulací s privilegia, stejně jako použitím principu nejmenšího privilegia rozhodnout, kdy zrušit systémová oprávnění. Je to také důležité rozdělit systém na bezpečné oblasti, které mají jednoznačné hranice důvěry, nedovolit citlivým údajům opustit hranice důvěry a opatrně komunikovat mimo jejich hranice.

2.6 Příklad chybné správy zdrojů: Privoxy

2.6.1 Chyba v programu

Privoxy je oblíbený webový proxy server bez ukládání do mezipaměti s pokročilými možnostmi filtrování pro zvýšení soukromí, úpravu dat webových stránek a záhlaví HTTP, řízení přístupu a odstraňování reklam a dalšího nepříjemného internetového odpadu. Privoxy má flexibilní konfiguraci a lze ji přizpůsobit tak, aby vyhovovala individuálním potřebám a vkusu. Má aplikaci jak pro samostatné systémy, tak pro víceuživatelské sítě. Jde o free software stažitelný z https://www.privoxy.org/sf-download-mirror/. Privoxy však umožňoval vzdáleným útočníkům způsobit odmítnutí služby prostřednictvím nespecifikovaných vektorů.

Toto selhání lze vidět ve fragmentu kódu na obrázku 2.6.2. Existuje blok části funkce, která interpretuje regulární výrazy. Problém s kódem je, že „l“ v prvním bloku může překročit počet alokovaných náhrad a chyba by skončila nastavením návratového kódu na chybu a začaly by se shromažďovat chyby, které by se nikdy neuvolnily.

```
/* Backreferences */
if (replacement[i] == '$' && !quoted && i < (int)(length - 1))
{
    char *symbol, symbols[] = "'`+&";
    r->block_length[l] = (size_t)(k - r->block_offset[l]);

    /* Numerical backreferences */
    if (isdigit((int)replacement[i + 1]))
    {
        while (i < (int)length && isdigit((int)replacement[++i]))
        {
            r->backref[l] = r->backref[l] * 10 + replacement[i] - 48;
        }
        if (r->backref[l] > capturecount)
        {
            *errptr = PCRS_WARN_BADREF;
        }
    }
```

Obrázek 2.6.1: Příklad selhání Privoxy

2.6.2 Oprava

Naštěstí, jak je vidět na obrázku 2.6.2, je tato oprava poměrně jednoduchá.

Nejprve zkontrolujte, zda funkce nepřekračuje počet výměn. A ujistěte se, že jste okamžitě uvolnili všechny zdroje, abyste v programu neshromažďovali zbytečné chyby.

Obrázek 2.6.2: Příklad opravy Privoxy

Existuje mnoho různých způsobů, jak opravit chyby správy prostředků v průběhu procesu vývoje kódu. Statická analýza například odhalí mnohé věcí, které mohou ztěžovat odstraňování chyb. Ruční analýza může samozřejmě také pomoci aktivním prozkoumáním všech zdrojů, ale je třeba mít na paměti, že je to časově náročné a obtížné, takže je nejlepší použít nástroj.

Je třeba nezapomenout na designové prvky. Zvážit přidání wrapperů[12] C++, které zabraňují zneužití nebo zavěšení zdrojů, a prozkoumejte předpoklady o zdroji, limity a rovnováhy a provádět opravy jednoduchým vytvořením lepšího kódu.

2.7 Příklad zranitelnosti kódu: CHRONY (NTP)

2.7.1 Chyby v programu

Příklad demonstrující tuto sbírku zranitelností, je chrony (NTP)[13], kde se neinicializuje poslední „další“ ukazatel při ukládání nepotvrzených odpovědí na požadavky příkazů. Co se nakonec stane, je spuštění libovolného kódu nebo alespoň umožňuje vytvoření útoku odmítnutí služby.

Na obrázku 5 je úryvek kódu vysvětlující zranitelnost. Zobrazený vzorek je velmi jemný a může ve skutečnosti vyžadovat určité studium, než programátor zjistí, proč existuje zranitelnost. Problémem v příkladu je v zásadě to,

[12] Knihovny wrapperů se skládají z tenké vrstvy kódu (shim), která převádí existující rozhraní knihovny do kompatibilního rozhraní. Řeší se tím problém interoperability mezi jazyky, nekompatibilních datových formátů, špatně navržených rozhraní atd.

[13] chrony je implementace protokolu NTP (Network Time Protocol). Může synchronizovat systémové hodiny s NTP servery, referenčními hodinami (např. přijímač GPS) a manuálním vstupem pomocí náramkových hodinek a klávesnice. Může také fungovat jako server NTPv4 (RFC 5905) a peer poskytovat časovou službu ostatním počítačům v síti.

že se jedná o velmi dlouhý a komplikovaný test a že logické OR v kódu způsobí neplatné časové razítko, když dojde k issue_token = 1. To znamená, že bude povoleno vydání tokenu bez řádného prozkoumání požadavku.

```
valid_ts = 0;

if (auth_ok) {
    struct timeval ts;

    UTI_TimevalNetworkToHost(&rx_message.data.logon.ts, &ts);
    if ((utoken_ok && token_ok) ||
        ((ntohl(rx_message.utoken) == SPECIAL_UTOKEN) &&
         (rx_command == REQ_LOGON) &&
         (valid_ts = ts_is_unique_and_not_stale(&ts, &now))))
        issue_token = 1;
    else
        issue_token = 0;
} else {
    issue_token = 0;
}
```

Obrázek 2.7.1: Příklad selhání chrony

2.7.2 Oprava

Oprava na obrázku 2.7.2 v zásadě dělá logiku ovládání mnohem lépe viditelnou a mnohem jasnější. To je způsob, jakým by měl být kód vždy navržen od začátku, aby se kód nestal příliš komplikovaným. I když se to může zdát nudné, někdy kódování podle příručky zachrání vývojáře před zranitelností.

```
valid_ts = 0;
issue_token = 0;

if (auth_ok) {
    if (utoken_ok && token_ok) {
        issue_token = 1;
    } else if (rx_command == REQ_LOGON &&
               ntohl(rx_message.utoken) == SPECIAL_UTOKEN) {
        struct timeval ts;

        UTI_TimevalNetworkToHost(&rx_message.data.logon.ts, &ts);
        valid_ts = ts_is_unique_and_not_stale(&ts, &now);

        if (valid_ts) {
            issue_token = 1;
        }
    }
}
```

Obrázek 2.7.2: Příklad opravy

Ujistěte se, že používáte dobře identifikované vzory kódování, protože složité kódování nikomu nedělá dobře. Vytvářejte také konzistentní kontrakty API a sledujte používání a ukládání hesel šifrovaných dat.

Dávejte pozor na nečistý kód, protože často poukazuje na špatnou kvalitu a potenciální bezpečnostní problémy. Nečistý kód sám o sobě není vždy zranitelností, ale nečistým kódem může být indikátorem toho, že někde neproběhly metodické procesy. A nezapomeňte zvládnout všechny chyby, aby nedošlo k nějakému překvapení.

Dobré kódování znamená dobré zabezpečení. Příručky stylů kódování nemusí být nejoblíbenější, ale buďte si jisti, že existují, aby udrželi zranitelnosti na uzdě.

2.8 Příklad zranitelnosti kódu: Injekce kódu

2.8.1 Chyby v programu

Jedním z příkladů je Windows RT, což je zajímavé, protože se nejedná o open zdrojový kód. To dokazuje, že i když může být něco proprietární, neznamená to, že není vystaveno zranitelnosti. To je důležité vědět při integraci těchto technologií do kódu. Něco tak nevinného zdánlivě jako schopnost zobrazit fonty písma může být důvodem, proč se kód stal zranitelným.

2.8.2 Oprava

Kontrola návrhu, manuální analýza a automatizovaná statická analýza mohou napomoci napravit zranitelná místa, ale programátor si musí uvědomit, že zabezpečení proprietárního kódu může být náročné.

Z hlediska designu si vývojáři musí být vědomi všech součástí černé skříňky a toho, že všechny jsou aktuální, a neměli by předpokládat, že vše je v interním systému. Je důležité, aby vývojáři věnovali zvláštní pozornost položkám, které budou stahovat z webu.

Statická analýza kódu (SCA – Static Code Analysis) může v této oblasti pomoci tím, že identifikuje použití nebezpečných dat, která procházejí systémem. To je místo, kde se vývojáři pravděpodobně zachytí útok založený na injekci. Ruční čištění jakýchkoli externě získaných informací také pomůže udržet systémy v bezpečí.

Je důležité mít na paměti veškerý kód, který přichází do ekosystému; předpokládejme, že v několika fázích životního cyklu vývoje softwaru existují zranitelná místa.

2.9 Příklad kryptografických problémů: Použití OPENSSL

2.9.1 Chyby v programu

OpenSSL se před lety proslavil Heartbleed[14]. Dalším problémem s OpenSSL je, když nezajistí, že pseudonáhodné generátor čísel se nasadí, než bude pokračovat v handshake.

[14] Heartbleed byla bezpečnostní chyba v kryptografické knihovně OpenSSL, což je široce používaná implementace protokolu Transport Layer Security (TLS). Zavedena byla v roce 2012 a zveřejněna v roce 2014.

To v podstatě znamená, že není dostatek informací, které by zajistily, že výstup není předvídatelný, když prochází algoritmem. To znamená, že prostor, který je potřeba k testování možných klíčů proti zašifrovaným datům, je drasticky snížen a útočníci tak budou nakonec schopni šifrování prolomit.

Hackování šifrování Wi-Fi WPA je dobrým příkladem, protože způsob, jakým jsou tato čísla nasazena, bez ohledu na zabezpečení WPA, umožňuje hacknutí kvůli konzistentní době kolem spuštění.

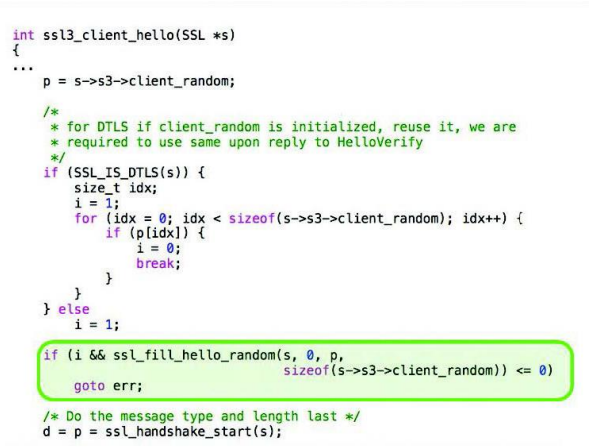
Jak je vidět na obrázku 2.9.1, dotyčný příkaz je částí klientské vyrovnávací paměti ve smyčce, kde říká, že pokud nastane podmínka, vyplní náhodnou vyrovnávací paměť. Problém je v tom, že fill_hello_random není voláno, pokud je podmínka je nenulová. To znamená, že tímto kódem existují cesty, kde při zahájení handshake nebude náhodná vyrovnávací paměť, což může vést k předvídatelným výsledkům.

```
int ssl3_client_hello(SSL *s)
{
...
    p = s->s3->client_random;

    /*
     * for DTLS if client_random is initialized, reuse it, we are
     * required to use same upon reply to HelloVerify
     */
    if (SSL_IS_DTLS(s)) {
         size_t idx;
         i = 1;
         for (idx = 0; idx < sizeof(s->s3->client_random); idx++) {
              if (p[idx]) {
                   i = 0;
                   break;
              }
         }
     } else
         i = 1;

    if (i)
         ssl_fill_hello_random(s, 0, p, sizeof(s->s3->client_random));

    /* Do the message type and length last */
     d = p = ssl_handshake_start(s);
```

Obrázek 2.9.1: Příklad selhání OpenSSL

2.9.2 Oprava: vždy volat funkci náhodně

Jak je vidět na obrázku 2.9.2, vždy by měl být proveden test, aby se ujistil, že hodnota je přijatelná, a poté by měla být vždy volána randomizační funkce. Takže když se objeví nulový příznak, může být zrušen, což zajišťuje, že handshake nemůže začít bez randomizační vyrovnávací paměti.



```
int ssl3_client_hello(SSL *s)
{
...
    p = s->s3->client_random;

    /*
     * for DTLS if client_random is initialized, reuse it, we are
     * required to use same upon reply to HelloVerify
     */
    if (SSL_IS_DTLS(s)) {
        size_t idx;
        i = 1;
        for (idx = 0; idx < sizeof(s->s3->client_random); idx++) {
            if (p[idx]) {
                i = 0;
                break;
            }
        }
    } else
        i = 1;

    if (i && ssl_fill_hello_random(s, 0, p,
                                   sizeof(s->s3->client_random)) <= 0)
        goto err;

    /* Do the message type and length last */
    d = p = ssl_handshake_start(s);
```

Obrázek 2.9.2: Příklad opravy OpenSSL

2.10 Příklad numerické chyby: QT[15]

2.10.1 Chyby v programu

Dokonalým příkladem numerické chyby je chyba v bitmapovém dekodéru nebo dekodéru souborů BMP v Qt. Jak je vidět na obrázku 2.10.1, kód správně nevypočítá masky použité k extrakci barevných složek, jako jsou červené, zelené, modré a alfa složky obrázku.

```
} else if (comp == BMP_BITFIELDS && (nbits == 16 || nbits == 32)) {
    red_shift = calc_shift(red_mask);
    red_scale = 256 / ((red_mask >> red_shift) + 1);

    green_shift = calc_shift(green_mask);
    green_scale = 256 / ((green_mask >> green_shift) + 1);

    blue_shift = calc_shift(blue_mask);
    blue_scale = 256 / ((blue_mask >> blue_shift) + 1);

    alpha_shift = calc_shift(alpha_mask);
    alpha_scale = 256 / ((alpha_mask >> alpha_shift) + 1);

} else if (comp == BMP_RGB && (nbits == 24 || nbits == 32)) {
```

Obrázek 2.10.1: Příklad selhání Qt

Jak ukazuje kód na obrázku 2.10.1, existují hodnoty posunu a měřítka, které se vypočítávají z bitmapy, která se načítá, a v tomto příkladu se 256 dělí vypočítanou hodnotou barevné masky posunuté o hodnotu posunu a pak jednu přidat. To způsobí zhroucení systému, pokud je hodnota masky je spousta jedniček s a hodnota posunu je spousta nul, protože 256 nelze vydělit nulou.

[15] Qt (Questions tagged) je multiplatformní aplikační vývojový rámec sloužící pro vývoj aplikačního softwaru, který lze provozovat na různých softwarových a hardwarových platformách s malou nebo žádnou změnou v základní kódové základně, přičemž má sílu a rychlost nativních aplikací. Blíže (Qt manul) anebo (Qt kurz).

I když někteří lidé mohou tvrdit, že tato situace nikdy nenastane, o to nejde. Hlavním bodem je, že zabezpečení je velmi obtížné, protože nejde jen o správné kódování, ale také o zlomyslné myšlení a defenzivní kódování. Už to není přijatelné jednoduše kód pro dobré okolnosti, musí nyní vývojáři kódovat defenzivně pro špatné okolnosti.

2.10.2 Oprava

Obrázek 2.10.2 ukazuje, jak vyřešit zranitelnost v Qt přidáním kódu, který kontroluje pokaždé, když jmenovatel může vyjít jako rovný nule – obrázek je označen jako špatný a program přeskočí tuto část kódu. To zajistí, že kód nikdy nevygeneruje nulovou odpověď, protože kód je pokaždé testován, aby bylo zajištěno, že program vyřeší špatné obrázky.

```
} else if (comp == BMP_BITFIELDS && (nbits == 16 || nbits == 32)
    red_shift = calc_shift(red_mask);
    if (((red_mask >> red_shift) + 1) == 0)
        return false;
    red_scale = 256 / ((red_mask >> red_shift) + 1);
    green_shift = calc_shift(green_mask);
    if (((green_mask >> green_shift) + 1) == 0)
        return false;
    green_scale = 256 / ((green_mask >> green_shift) + 1);
    blue_shift = calc_shift(blue_mask);
    if (((blue_mask >> blue_shift) + 1) == 0)
        return false;
    blue_scale = 256 / ((blue_mask >> blue_shift) + 1);
    alpha_shift = calc_shift(alpha_mask);
    if (((alpha_mask >> alpha_shift) + 1) == 0)
        return false;
    alpha_scale = 256 / ((alpha_mask >> alpha_shift) + 1);
} else if (comp == BMP_RGB && (nbits == 24 || nbits == 32)) {
```

Obrázek 2.10.2: Příklad opravy Qt

Je nemožné si skutečně být jisti, odkud obrázky přicházejí. Pokud je špatný obrázek vytvořen správným způsobem, dojde k selhání softwaru, který tento obrázek vytváří. Často se předpokládá, že loga a obrázky pocházejí z bezpečného místa, ale kód by mohl stahovat obrázky z Bluetooth nebo webových aplikací, aniž by to vývojáři viděli.

Shrnutí a závěr

Chcete-li pomoci s bezpečnostními chybami v programech, měli byste vynucovat, regulovat a věnovat pozornost:

- Směrnice pro architekturu a design.
- Prováděcí pokyny.
- Manuální analýza.
- Automatická statická analýza.
- Automatická dynamická analýza.

Bezpečnost je třeba považovat za vysokou prioritu během celé fáze budování. Je důležité si uvědomit, že bez ohledu na to, jak malá nebo bezvýznamná chyba vypadá, může vést ke škodlivému útoku.

Existují čtyři osvědčené postupy k zajištění ochrany před běžnými zranitelnostmi: čistý design, metodický proces, pečlivá analýza a dobré nástroje.

Čistý design: Vytvoří se návrh, který čistě odděluje procesy s různými potřebami zabezpečení a který má přísně kontrolované rozhraní mezi komponentami. Neposkytujte více přístupu k systémovým komponentám, než je nezbytně nutné. Používají se pouze osvědčené a skutečné technologie k ověření, šifrování a zabezpečení daného produktu. V případě pochyb programátor konzultuje s bezpečnostním specialistou.

Metodický proces: Procesy fungují pouze tehdy, když jsou dodržovány. Vědomí, že procesy mohou zachytit hloupé chyby, nečiní vývojáře nedbalým – pouze se zvyšuje jeho sebevědomí.

Pečlivá analýza: Je si třeba pečlivě promyslete každé místo, kde se hacker může k systému dostat. Programátor by se neměl nechat uvěznit konvenčním myšlením o navržených existujících rozhraních. Pečlivě by měl prozkoumat každé rozhraní a ujistit se se, že je co nejodolnější. Zkoumat rovněž, kde program spotřebovává data, neberte nic jako samozřejmost.

Dobré nástroje: Mnoho chyb nalezených analýzou statického kódu se přímo promítne do uzavření zranitelností. Použití nástrojů k vyčištění nedbalých praktik kódování bude mít za následek přísnější a bezpečnější kód. A mnoho nástrojů má speciální konfigurace navržené tak, aby hledaly konkrétní bezpečnostní problémy.

Je třeba zajistit **výběr vhodného nástroje pro kontrolu zranitelností kódu** a ten by měl splňovat:

1. Dodržování standardů

Nástroj by se měl řídit standardy a pokyny pro bezpečné kódování: CERT, CWE, OWASP, DISA STIG atd.

2. Použití sady pomocných nástrojů (vývojářský desktop, testování před sloučením a po sloučení) v každém kontrolním bodě. Tyto nástroje umožňují vývojářům:

- Nalézt defekty při psaní kódu.
- Kontroly provádět v čistějším kódu.
- Definujte QA (Quality Assurance – zajištění jakosti), bezpečnostní cíle a konfigurace pravidel.

- Vytváření bezpečnostní zprávy.
- Upřednostnění defektů na základě závažnosti, umístění a životního cyklu.
- Upřednostnění oprav na základě pravděpodobnosti defektu, což v kombinaci se závažností problému poskytuje celkové skóre rizika zranitelnosti.
- Rozlišení nových problémů od problémů se starším kódem.

3. Diferenciální analýza

Diferenciální analýza je formou statické analýzy „rychlé zpětné vazby“, která využívá kontextová data systému z předchozích sestavení analýzy k analýze pouze nových a změněných souborů. Tento typ analýzy poskytuje vývojářům nejkratší možné doby analýzy pro nový a změněný kód a zároveň zachovává přesnost a podrobnost analytických dat. Namísto čekání hodin mohou vývojáři dostat výsledky v minutách nebo sekundách v závislosti na tom, jak moc se kód změnil.

4. Analýza toku dat

Nejobtížněji se hledají problémy, kdy přechází toky dat mezi funkcemi a přes hranice souborů. Je třeba sledovat data, jak proudí mezi metodami, soubory a moduly, aby našel zranitelná místa, jako je použití poškozených nebo neinicializovaných dat.

5. Vytvoření vlastních pravidel

To umožňuje vývojářům upozorňovat na nebezpečné praktiky, které jsou jedinečné pro jejich vlastní kódové základny.

Kontrolu bezpečnosti je třeba provádět jak ve fázi jeho sestavení, tak ve fázi jeho ověřování.

Sestavení: Je třeba nástroje integrovat jako součást procesu sestavení od samého začátku, aby bylo možné využít výhod konzistentních kontrol v přírůstkových denních sestavách, což zajišťuje, že vkládání nástrojů do vyzrálého procesu sestavování není úplně tak skličující úkol. Není vhodné čekat na konec projektu, kdy vysoká setrvačnost a riziko zavlečení chyb povede k odhlášení z mnoha malých oprav, které mohou nástroje indikovat.

Ověření: Ověření závisí na vstupu nejen z interního kódu, ale také z kódu, na který se spoléhá. Najděte nástroje, které mohou pomoci toku kontrol integrity během celého integračního procesu. A nepřestávejte spouštět nástroje jen proto, že je program ve výrobě. Spolehněte se na ně, že po zlatém sestavení znovu zkontrolují opravy chyb nebo doplnění funkcí, abyste se ujistili, že jakýkoli nový přidaný kód dodržuje stejnou výchozí laťku vysoké kvality.

Při vývoji je třeba vzít v úvahu aspekty jako nástroje pro písma, webové prohlížeče, prohlížeče PDF, nástroje pro rozpoznávání řeči, sady grafických nástrojů a Adobe Flash a AIR. To, že tento kód nebyl vytvořen interně, neznamená, že nemůže být hacknut – je třeba ověřit všechny vstupy, které naprogramované moduly přijímají, nebo jak tyto komponenty co nejvíce izolovat od zbytku systému, aby neúmyslné narušení bezpečnosti se nemohlo rozšířit daleko.

Softwarové procesy se musí vytvářet za předpokladu, že výroba nezastaví vývoj softwaru a že systém bude potřebovat technologii a infrastrukturu, aby mohl přijímat aktualizace.

Povědomí o typových problémech může pomoci s téměř 90 % všech zranitelností ve vestavěném softwaru. Zbývajících 10 % zranitelností lze odstranit spoléháním na správné nástroje, dobrý design, dobře promyšlené procesy a pečlivá analýzu.

Seznam použitých zdrojů

(Ashley, 2021) ASHLEY, Mitch. Top 10 Embedded Security Vulnerabilities. DevOps.com. January 11, 2021. Dostupné z: https://devops.com/top-10-embedded-security-vulnerabilities/

(Buchanan, 2022) Prof Bill Buchanan OBE. A Long Goodbye to C and C++, And Hello To A Rusty Future, Sep 24 2022 Dostupné z: https://medium.com/asecuritysite-when-bob-met-alice/goodbye-to-c-and-c-and-hello-to-a-rusty-future-10da003dee5e

(Qt manual) Qt Creator Manual. Dostupné z: https://doc.qt.io/qtcreator/index.html

(Qt kurz) Qt - Okenní/formulářové aplikace v C++ - Online kurz. Dostupné z: https://www.itnetwork.cz/cplusplus/qt

(Perforce 2021a) The Best Static Analysis and SAST Tool for Accelerating Time-to-Market and Delivering High Quality, Secure, and Compliant Code. Dostupné z: https://www.perforce.com/sites/default/files/pdfs/datasheet-klocwork-sast.pdf

(Perforce 2021b) Top 10 Embedded Software Cybersecurity Vulnerabilities. Perforce 2021. Dostupné z: www.perforce.com/blog/kw/common-software-vulnerabilities

(Perforce 2021c) A Guide to Security in Software Development. Perforce Software 2021. Dostupné z: https://www.perforce.com/resources/kw/secure-coding-standards-guide?utm_term=trendemon