Biztonságos programok készítése

Az új kódok, amiket te írtál, bizonyára biztonságosak, de mi legyen az olyan nagyobb tervezetekkel, amit mástól kell átvenned?

A biztonság kapcsán két fő terület jut eszünkbe, a hálózati és a fizikai biztonság. A biztonságos programok írására sokkal kevesebb figyelem jut. A biztonságos programok készítéséhez szükséges eljárások alkalmazása nem lenne ördöngösség, ám a szűkös idő és a tervezési szakasz lerövidítése miatt ritkán sikerül követni őket. Minden jó programozó ismeri az elveket, de nincs ideje alkalmazni őket – túl nagy nyomás nehezedik rá: nagymennyiségű kódot kell elkészítenie, a tervezettel pedig muszáj időben végezni.

Az 1970-es évek elején virágkorát élte a strukturált programozás. Akkoriban nemcsak a program, de az egész tervezet gondosan összeállított szerkezettel bírt, megvoltak a szükséges műszaki elvárások, részletes tervezési előírások, valamint a tervezést és a kódot is gondosan végigrágták. Természetesen a tervezetek nagyobbak lettek, befejezésükhöz több időre volt szükség, ám amikor elkészültek, a kód hibáinak megkeresése könnyen ment, és sokszor csak apróbb módosításokra volt szükség, hogy tökéletesen működjenek. Némelyik tervezet befejezése akkoriban több évig is eltartott.

Napjainkra a strukturált fejlesztési eljárások túlnyomó része eltűnt. Csakhogy a biztonság megteremtése a program tervezésénél kezdődik, és a munka színhelyéül szolgáló szervezet által felállított programírási szabványoktól függ. Senki nem gondol arra, hogy a kódja százszázalékosan biztonságos lesz, ugyanis ilyen kód nincs is. Arra azonban törekedni kell, hogy a kód üzembiztos és biztonságos legyen. Mit lehet tenni ennek érdekében? Írásomban néhány ötletet és három biztonságos programírást segítő eszközt szeretnék ismertetni. Beágyazott rendszerek tervezésekor és megvalósításakor különös gondossággal kell eljárni. A biztonságos programok írása végső soron a kód készítőjének feladata, az ő képességeinek függvénye, hogy megérti-e mit is kezelhet biztonságosként.

Hibakeresés

Minden függvény visszaad valamilyen típusú állapotjelzést, akár közvetlenül, akár az `errno` kimeneten keresztül. Ezek ellenőrzése egyszerű. A C++ alatt a kivételkezelés könnyen használható, ám annál nehezebb lehet a beállítása. Az utóbbi évek során, a C++ szabvány befejezése óta a kivételkezelés rengeteget fejlődött. Ha lehetséges, használni is kell őket.

A korábbi gyakorlat szerint a hibákat figyelmen kívül hagyták, mert mindenki úgy gondolta, maguk az átadott adatok helyesek. Mint később bebizonyosodott, ez a hozzáállás hibás. Az adat-átmenetitárak túlcsordulásai (data buffer overflows) miatt az elmúlt években rengeteg hibát kellett kijavítani. Ha beágyazott rendszerre fejlesztünk, a hibajelzések áttekintése nem mellőzhető. Ezután még mindig dönthetünk úgy, hogy a hiba jóindulatú, és ezért figyelmen kívül hagyhatjuk. Ha mégsem, meg kell kísérelni a hiba kijavítását. Amennyiben erre nincs mód, a rendszer alapállapotba állítását vagy a gép újraindulását kell eredményeznie. Sok esetben elegendő a program alapállapotba állítása is, ennek során a hibához vezető művelet újrakezdődik. Ez minden hibatűrő rendszer alapja. Az adott készülék típusától függően a gép újraindítása is megfelelő és elegendő megoldás lehet, míg akadnak olyan esetek, amikor elengedhetetlen valamilyen helyreállítási lehetőség biztosítása.

Karakterláncok kezelése

Az `sprintf`, az `strcpy` és az `strcat` helyett inkább az `strncpy` és az `strncat` függvényeket kell használni. Ezek szavatolják, hogy az átmeneti tár nem fog túlcsordulni, a túlnyúló részeket pedig elhagyják a karakterláncokból. Az `fgets` függvényt adatok beolvasására ne alkalmazzuk, mert átmenetitár-túlcsordulást okozhatunk vele. Ezek egészen egyszerű módosításoknak tűnnek, de könnyű róluk megfeledkezni, és éppen ezért a legtöbb kihasználható hiba forrása a helytelen karakterlánc-kezelés a programokban. Az önműködő ellenőrző programok ugyan elég jól szűrik az ilyen hibákat, de sokszor félrevezethetik használójukat. Gyakorta egy-egy karakterlánc-kezelő függvény használatát hibásnak minősítik, annak ellenére, hogy az adott környezetben a programozó megfelelő megoldást választott. Ezek azok a helyzetek, amikor a programozó képességei és tudása kerülnek előtérbe, ugyanis az ő feladata, hogy az ellenőrzőprogram naplóját átvizsgálja és megállapítsa, hogy mely esetekben szükséges módosításokat végezni.

Memóriaszivárgások és átmenetitár-túlcsordulások

A memóriaszivárgások önmagukban nem feltétlenül jelentenek biztonsági kockázatot. Ha viszont a memóriaterület több eljárás és adatszerkezet között van megosztva, kiaknázható hibát jelenthetnek.

A átmenetitár-túlcsordulások messze a leggyakoribb biztonsági kockázatforrások. Ha egy átmeneti tár a veremben található, akkor túlcsordulását előidézve törölhető vagy megváltoztatható egy függvény visszatérési címe. Amikor a függvényhívás visszatér, az eredeti helyett immár az új címmel teszi meg. Bizonyos átmenetitár-támadásokat (buffer attacks) a kupacon (heap) is végre lehet hajtani. Igaz, hogy egy ilyen támadást nehezebben lehet kivitelezni, de egyáltalán nem lehetetlen. A C nyelven írt programok sebezhetőbbek az ilyen támadásokkal szemben, de gyakorlatilag minden alacsonyszintű memóriakezelési és mutatóhasználati lehetőséget kínáló nyelv esetében jelentkezhetnek gondok. A mutatókezelés jó példa azokra a területekre, amelyeken nem szabad elfeledkezni a határok ellenőrzéséről.
A GNU C fordítóhoz létezik olyan kiterjesztés – a fordító fordításakor kell beépíteni –, amely a határok ellenőrzését elvégzi. Kiegészítő jelleggel használható, a programot külön kódrészekkel bővíti. A kipróbálás időszakára ezek a részek engedélyezhetők és használhatók, majd az alkalmazás telepítésekor le kell tiltani őket. Ennek oka az, hogy ezek a kódrészek a határok átlépésekor üzeneteket jelenítenek meg a képernyőn. Ha a program egy munkaállomáson fut, az üzenetek nem okoznak gondot, de egy beágyazott rendszeren, ahol általában nincs konzol, értelmüket vesztik.
E téma kapcsán felvetődhet az átmeneti tárak statikus lefoglalásának ötlete, ezzel a hibalehetőség azonnal megszűnne. Sajnos a szabott méretű átmeneti tárak jelenléte is kihasználható, a beléjük másolt adatok mérete ugyanis továbbra is nagyobb lehet az átmeneti tárénál. Ha az adatmásolás megtörténik, ugyancsak túlcsordulásra kerülhet sor. A kockázat csak olyan módon csökkenthető, ha az átmásolt adatok mérete nem haladhatja meg az átmeneti tárét. A karakterláncok helyét dinamikusan lefoglalva a programok tetszőleges méretű bemeneteket képesek kezelni. A gond ekkor csak az, hogy a program kifuthat a szabad memóriából. Beágyazott rendszeren az ilyen hiba végzetes lehet, munkaállomáson pedig a virtuális memóriarendszer kezdhet dolgozni serényen, ami viszont a számítógép teljesítményét veti vissza. C++ alatt az `std::string` osztály alkalmas a dinamikus szemlélet gyakorlatba ültetésére. Ha az osztály adatait egy `char *` mutatóval kezeljük, átmenetitár-túlcsordulás léphet fel. Az egyéb karakterlánc-kezelő könyvtáraknál ilyesmi nem fordulhat elő, de a programozónak mindvégig tekintettel kell lennie e korlátokra.

A bemenet ellenőrzése

Ha egy program kívülről kapja a további működése alapjául szolgáló adatokat, érdemes ellenőrizni, hogy az adatok mérete nem haladja-e meg a megengedettet, helyesek-e és nem tartalmaznak-e meg nem engedett típusú elemeket. Például, ha a bevitt adatsor csak A és Z közötti nagybetűkből állhat, akkor a függvénynek minden mást vissza kell utasítania. Ugyancsak ellenőrizni kell, hogy az adatok hossza érvényes-e. Nem is olyan rég volt, amikor az adat fogalma alatt mindenki 80 karaktert értett, azaz egy lyukkártya tárolási méretét. Manapság az adatmennyiség szinte tetszőleges lehet, típusára nézve pedig szöveges, bináris és titkosított egyaránt előfordulhat. Határok azonban még ma is léteznek. Az adatok ellenőrzését el kell végezni, és ha az eredmény nem megfelelő, a bemenetet vissza kell utasítani.
Sokszor nem elég a bemenet méretét a felső határhoz viszonyítani, meg kell vizsgálni azt is, hogy van-e olyan hosszú, mint amit alsó határként megszabtunk. A karakterláncokat abból a szempontból is ellenőrizni kell, hogy érvényes értékeket és karaktersorozatokat tartalmaznak-e. Ha az ellenőrzés alá vont adatok bináris formátumúak, és valamiért így is kell maradniuk, akkor a legjobb valamilyen általános kilépési karaktert használni a bináris formátum jelzésére. Ha szám típusú adatokkal dolgozunk, az értéktartomány betartására kell ügyelnünk. Ha pozitív egész számot várunk, mindig ellenőrizzük, hogy nullánál nem kisebbet kaptunk-e. Ha van valamilyen legnagyobb érték, ellenőrizzük, hogy nem léptük-e át. A ***limits.h*** fájlban a legtöbb típus maximum- és minimumértéke meg van adva, így ennek segítségével könnyedén ellenőrizhető a rendszer határértékeinek betartása.

Segédeszközök

A legtöbb fejlesztő került már olyan helyzetbe, hogy készen állt a kód, de csak nagyon kevés idő és energia maradt a lehetséges biztonsági hiányosságok felkutatására. Végül is a kód működik, miért kellene javítani rajta? Nagyon sok helyen tartja magát ez a felfogás. Ha mégis fény derül a kód valamilyen hibájára, azonnal fontos lesz a kijavítása, miként a felelőskeresgélés sem maradhat el.
Mit lehet tenni, ha rövid idő alatt kell meglelni a lehetséges hibaforrásokat? Három olyan eszközt említenék meg, amelyek képesek a lehetséges hibák felismerésére és jelentésbe gyűjtésére. Beágyazott rendszereken is használhatók, bár a valóságban a legtöbb fejlesztői munka vegyes környezetben zajlik. A munka nagy részét a munkaállomásokon szokták elvégezni, a célrendszeren már csak a finomhangolást szükséges végrehajtani.
A Flawfinder, a RATS és az ITS4 három olyan csomag, amely a forrásfát végigvizsgálja és a lehetséges hibákat jelenti. Kimenetük a forrásmodult és a kódsort is pontosan megjelölő hibalista. A hibajelzésekhez súlyozás is társul, amellyel a sebezhetőség mértékét adják meg.
Kódrészletünkben egy mintakódon futtatott Flawfinder kimenetének részlete látható. A hibák komolyságát 0 és 5 közötti számokkal jellemzi, ahol a 0 a csekély, az 5 pedig a komoly kockázatot jelenti.
A program akár jelentős számú hibát is jelezhet, ezt követően már a fejlesztő döntésén múlik, hogy kijavítja vagy figyelmen kívül hagyja-e őket. Sok fejlesztő tart tőle, hogy ezek a segédeszközök módosítsák is a kódot, de gondoljunk bele, még mindig jobb a módosítandó részeket kiválogatni, mint tömeges változtatásokat segédeszköz nélkül elvégezni. A Flawfinder program saját adatbázist és szabályhalmazt alkalmaz, ez a leggyakoribb biztonsági hiányosságok listáját tartalmazza.

Összegzés

Biztonságos kódot írni könnyű. Annak mérlegelése, hogy milyen szolgáltatásokat kell megvalósítani, és ezek hogyan használhatók támadások kivitelezésére, még a tervezési időszakban szükséges eldönteni és megvizsgálni. Az ellenőrzési eljárásoknak a különféle támadások és helytelen

használati módok kipróbálására is ki kell terjednie. Ezeket a próbákat teljesen önműködővé tenni fényűzés lenne, és a felhasználó is jóval később jutna hozzá a – kétségtelenül jobb minőségű – termékhez. Az itt említett módszerek és eszközök csak segítséget nyújtanak mindehhez. A programok biztonságának sorsa továbbra is a fejlesztők kezében marad, életútjuk az ő fejükben kezdődik.

Linux Journal 2003. november 115. szám

Cal Erickson (cal_erickson@mvista.com) jelenleg a MontaVista Software vezető Linux-tanácsadója. Mielőtt csatlakozott volna a MontaVista csapatához, a támogatást vezető mérnök volt a Mentor Graphics beágyazott alkalmazások fejlesztési részlegénél. Cal több mint harminc éve dolgozik a számítástechnikai iparágban, tapasztalatait számítógépgyártóknál és végfelhasználói termékeket fejlesztő cégeknél szerezte.

A Flawfinder kimenete

Flawfinder version 1.21, (C) 2001-2002 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 127 (A C/C++ szabályhalmazban található veszélyes eljárások száma: 127)

Examining ../../pelda_kod/msgqueue/mksem.c (Az mksem.c fájl vizsgálata.)
../../pelda_kod/msg_queue/msgtool.c:73 [4] (buffer) strcpy:
Does not check for buffer overflows when copying to destination. (Nem végez átmenetitár-túlcsordulás vizsgálatot a célba másolás alatt.)
Consider using strncpy or strlcpy (warning, strncpy is easily misused). (Fontolja meg az strncpy vagy az strlcpy használatát. Vigyázat, az strncpy használata könnyen elrontható!)
../../pelda_kod/msgqueue/mksem.c:34 [4] (shell) system:
This causes a new program to execute and is difficult to use safely. (Új program futtatását okozza, biztonságosságát nehéz garantálni.)
Try using a library call that implements the same functionality if available. (Próbáljon könyvtári hívást használni erre a célra, ha lehetséges.)
../../pelda_kod/pipes/fifo/fifo_out.c:28 [4] (race) access:
This usually indicates a security flaw. If an attacker can change anything along the path between the call to access() and the file´s actual use (e.g., by moving files), the attacker can exploit the race condition. (Általában biztonsági hiányosságot jelent. Ha a támadó bármit meg tud változtatni az access() hívás és a fájl tényleges használatának ideje között (például fájlok átmozgatásával), akkor kihasználhatja a versenyhelyzetet.) Set up the correct permissions (e.g., using setuid()) and try to open the file directly. (Állítson be helyes engedélyeket (például a setuid() segítségével), és próbálja meg közvetlenül megnyitni a fájlt.)
../../pelda_kod/process_control/proc_mem_info/proc_mem_info.c:139 [4] (buffer) sscanf:
The scanf() family´s %s operation, without a limit specification, permits buffer overflows. (A scanf() család %s formátumú hívása korlát nélkül alkalmazva átmenetitár-túlcsorduláshoz vezethet.) Specify a limit to %s, or use a different input function. (Adjon meg korlátot a %s formátumú híváshoz, vagy válasszon másik beviteli függvényt.)
../../pelda_kod/msg_queue/sender/snd_thread.c:70 [3] (random) srand:
This function is not sufficiently random for security-related functions such as key and nonce creation. (A függvény nem biztosít elegendő véletlenszerűséget ahhoz, hogy biztonsági eljárásokban – például kulcsok vagy egyszer használatos véletlen adatok előállításához – használjuk.) Use a more secure technique for acquiring random values. (A véletlen értékek előállításához használjon biztonságosabb módszert.)
../../pelda_kod/dlopen/dltest.c:30 [2] (misc) fopen:
Check when opening files – can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents? (Fájlok megnyitásakor ellenőrizze, vajon a támadó nem tudja-e átirányítani a műveletet – például közvetett hivatkozások segítségével –, nem tudja-e különleges fájl – például eszközfájl – megnyitására rávenni a rendszert, nincs-e módja versenyhelyzet létrehozására vagy a szülőelemek, vagy a tartalom módosítására.)
../../pelda_kod/msg_queue/receiver/rcvr.c:51 [2] (buffer) char:
Statically-sized arrays can be overflowed. (A statikus méretű tömbök túlcsordulhatnak.) Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length. (Végezzen határellenőrzést, használjon hosszkorlátozó függvényeket vagy ellenőrizze, hogy a tömb mérete nagyobb-e, mint a lehetséges leghosszabb bemeneté.)
../../pelda_kod/dlopen/another_dlopen_test/obj.c:15 [1] (buffer)
strlen:
Does not handle strings that are not \0-terminated (it could cause a crash if unprotected). (Nem kezeli a nem \0 végződésű karakterláncokat. Kellő védelem hiányában összeomlást okozhat.)

...

Number of hits = 139 (Találatok száma = 139)
Number of Lines Analyzed = 5491 in 2.67 seconds (2527 lines/second) (Elemzett sorok száma = 5491, idő: 2,67 másodperc (2527 sor/másodperc)
Not every hit is necessarily a security vulnerability. (Nem minden találat jelent feltétlenül biztonsági hiányosságot.)
There may be other security vulnerabilities; review your code! (Ellenőrizze a kódot, további biztonsági hibákat is tartalmazhat!)