Audyt kodu źródłowego: znajdowanie luk w językach opartych na C

Audyt oprogramowania za pomocą kodu źródłowego jest często najskuteczniejszym sposobem wykrywania nowych luk w zabezpieczeniach. Duża ilość szeroko stosowanego oprogramowania to oprogramowanie typu open source, a niektórzy komercyjni dostawcy udostępnili publicznie swój kod źródłowy systemu operacyjnego. Przy pewnym doświadczeniu możliwe jest szybkie wykrycie oczywistych wad i bardziej subtelnych wad z czasem. Chociaż analiza binarna jest zawsze możliwa, dostępność kodu źródłowego sprawia, że przeprowadzanie audytów jest dużo łatwiej. Ta Część obejmuje audyt kodu źródłowego napisanego w językach opartych na C pod kątem zarówno prostych, jak i subtelnych luk w zabezpieczeniach, a przede wszystkim koncentruje się na wykrywaniu luk w pamięci. Wiele osób kontroluje kod źródłowy i każdy ma swoje własne powody do robienia tego. Niektórzy audytują kod w ramach swojej pracy lub hobby, podczas gdy inni po prostu chcą mieć pojęcie o bezpieczeństwie aplikacji, które uruchamiają w swoich systemach. Bez wątpienia są ludzie, którzy przeprowadzają audyt kodu źródłowego, aby znaleźć sposoby na włamanie się do systemów. Bez względu na powód przeprowadzania audytu, przegląd kodu źródłowego jest prawdopodobnie najlepszym sposobem wykrywania luk w aplikacjach. Jeśli kod źródłowy jest dostępny, użyj go. Spór o to, czy trudniej jest znaleźć błędy lub je wykorzystać, został rzucony, a sprawy można postawić po obu stronach. Niektóre luki w zabezpieczeniach są niezwykle oczywiste dla każdego, kto czyta źródło, ale okazują się prawie nie do wykorzystania w każdej praktycznej sytuacji. Jednak odwrotność jest bardziej powszechna i z mojego doświadczenia wynika, że wąskim gardłem w badaniach podatności jest najczęściej wykrywanie podatności jakościowych, a nie ich wykorzystywanie. Niektóre luki są natychmiast rozpoznawalne i można je szybko wykryć. Inne są dość trudne do zauważenia, nawet jeśli zostaną ci wskazane. Różne pakiety oprogramowania oferują różne poziomy trudności i różne wyzwania. Bez wątpienia istnieje wiele źle napisanego oprogramowania, ale jednocześnie istnieje również bardzo bezpieczne oprogramowanie typu open source. Skuteczny audyt opiera się na uznaniu i zrozumieniu. Wiele luk w zabezpieczeniach wykrytych w różnych aplikacjach jest dość podobnych, a jeśli uda Ci się znaleźć lub rozpoznać lukę w jednej aplikacji, istnieje duża szansa, że ten sam błąd został popełniony w innym miejscu. Lokalizacje więcej subtelnych kwestii wymagają głębszego zrozumienia aplikacji i generalnie mają zakres znacznie większy niż jakakolwiek pojedyncza funkcja. Bardzo pomocna jest dogłębna znajomość audytowanej aplikacji. Szczerze mówiąc, obecnie o wiele więcej osób przeprowadza audyty kodu źródłowego niż miało to miejsce kilka lat temu, a wraz z upływem czasu zostaną znalezione bardziej oczywiste i łatwe do wykrycia błędy. Deweloperzy są coraz bardziej świadomi problemów związanych z bezpieczeństwem i rzadziej powtarzają błędy. Proste błędy, które trafiają do wydanego oprogramowania, są zwykle szybko zauważane i rzucane przez badaczy luk. Łatwo byłoby argumentować, że badanie podatności będzie tylko trudniejsze w miarę upływu czasu; jednak cały czas tworzony jest nowy kod, a nowe klasy błędów są od czasu do czasu odkrywane. Możesz polegać na fakcie, że luki w zabezpieczeniach pozostają w każdej większej aplikacji. To po prostu kwestia ich znalezienia.

Narzędzia

Audyt kodu źródłowego może być bolesnym zadaniem, jeśli jesteś uzbrojony tylko w edytor tekstu i grep. Na szczęście dostępne są bardzo przydatne narzędzia, które znacznie ułatwiają audyt kodu źródłowego. Ogólnie rzecz biorąc, narzędzia te zostały napisane w celu wspomagania rozwoju oprogramowania, ale działają równie dobrze w przypadku audytu. W przypadku małych aplikacji nie zawsze jest konieczne korzystanie ze specjalistycznych narzędzi, ale w przypadku większych aplikacji obejmujących wiele plików i katalogów narzędzia te stają się bardzo użyteczne.

Cscope

Cscope to narzędzie do przeglądania kodu źródłowego, które jest bardzo przydatne do inspekcji dużych drzew kodu źródłowego. Został pierwotnie opracowany w Bell Labs i został publicznie udostępniony na licencji BSD przez SCO.

Cbrowser

Wiele innych narzędzi oferuje podobną funkcjonalność do Cscope i Ctags. Na przykład Cbrowser oferuje graficzny interfejs do Cscope i może być przydatny dla osób, które przeprowadzają audyt w środowisku GUI.

Zautomatyzowane narzędzia do analizy kodu źródłowego

Istnieje kilka publicznie dostępnych narzędzi, które próbują przeprowadzić statyczną analizę kodu źródłowego i automatycznie wykryć luki. Większość z nich przydaje się jako punkt wyjścia dla początkującego audytora, ale żaden z nich nie doszedł do poziomu zastąpienia dokładnego audytu przez osobę doświadczoną. Wielu dużych dostawców oprogramowania używa wewnętrznych narzędzi do analizy statycznej do wykrywania prostych luk w zabezpieczeniach, zanim zostaną one wprowadzone do kodu produkcyjnego. Jednak braki tych narzędzi są oczywiste. Niemniej jednak mogą być przydatnym miejscem do szybkiego rozpoczęcia pracy z dużym i stosunkowo niezbadanym drzewem źródłowym. Splint to narzędzie do analizy statycznej przeznaczone do wykrywania problemów związanych z bezpieczeństwem w programach C. Dzięki adnotacjom dodanym do programów Splint może przeprowadzać stosunkowo silne kontrole bezpieczeństwa. W przeszłości wykazano, że silnik analizy automatycznie wykrywa problemy związane z bezpieczeństwem, takie jak przepełnienie BIND TSIG (choć już po tym, jak były już znane). Chociaż Splint ma problemy z dużymi i złożonymi drzewami źródłowymi, nadal warto się temu przyjrzeć. Został opracowany przez University of Virginia . CQual to aplikacja, która ocenia adnotacje dodane do kodu źródłowego C. Rozszerza standardowe kwalifikatory typu C o dodatkowe kwalifikatory, takie jak tainted, i ma logikę wnioskowania typu zmiennych, których kwalifikatory nie zostały wyraźnie zdefiniowane. CQual może wykryć pewne luki, takie jak ciągi formatujące; jednak nie znajdzie niektórych bardziej zaawansowanych problemów, które można wykryć za pomocą analizy ręcznej. CQual został napisany przez Jeffa Fostera . Inne narzędzia, takie jak RATS oferowane przez Secure Software, są dostępne, ale generalnie zostały zaprojektowane w celu zlokalizowania uproszczonych luk w zabezpieczeniach, które nie są często spotykane w nowoczesnym oprogramowaniu. Niektóre klasy błędów lepiej nadają się do wykrywania za pomocą analizy statycznej, a kilka innych publicznie dostępnych narzędzi automatycznie wykrywa potencjalne luki w ciągach formatu. Ogólnie rzecz biorąc, obecny zestaw narzędzi do analizy statycznej jest niewystarczający, jeśli chodzi o wykrywanie stosunkowo skomplikowanych luk w zabezpieczeniach współczesnego oprogramowania. Chociaż mogą one być dobre dla początkującego, najpoważniejsi audytorzy wykroczą daleko poza podzbiór luk, pod kątem których te programy mogą sprawdzić.

Metodologia

Czasami badacze bezpieczeństwa mogą mieć szczęście podczas audytu aplikacji bez przestrzegania żadnego konkretnego planu. Może się zdarzyć, że przeczytają właściwy fragment kodu we właściwym czasie i zobaczą coś, co wcześniej nie zostało zauważone. Jeśli jednak Twoim celem jest znalezienie określonej luki w konkretnej aplikacji lub próba znalezienia wszystkich błędów w jednej aplikacji (jak w przypadku każdego profesjonalnego audytu kodu źródłowego), potrzebna jest bardziej dobrze zdefiniowana metodologia. Jaka okaże się ta metodologia, zależy od twoich celów i typów luk, których szukasz. Poniżej przedstawiono kilka możliwych sposobów audytu kodu źródłowego.

Podejście odgórne (specyficzne)

W podejściu odgórnym do audytu kodu źródłowego audytor szuka określonych luk w zabezpieczeniach bez konieczności lepszego zrozumienia działania programu. Na przykład, audytor może przeszukać całe drzewo źródłowe w poszukiwaniu luk w ciągach formatu wpływających na funkcję syslog bez czytania programu linijka po linijce. Ta metoda może być oczywiście szybka, ponieważ audytor nie musi zdobywać dogłębnej wiedzy na temat aplikacji, ale taki audyt ma swoje wady. Wszelkie luki, które wymagają głębokiej znajomości kontekstu programu lub obejmują więcej niż jedną część kodu, prawdopodobnie zostaną pominięte. Niektóre luki można łatwo zlokalizować, po prostu czytając jeden wiersz kodu; to są rodzaje błędów, które będą znalezione metodą odgórną. Wszystko, co wymaga większego zrozumienia, będzie musiało zostać zlokalizowane w inny sposób.

Podejście oddolne

W podejściu oddolnym do audytu kodu źródłowego, audytor stara się uzyskać bardzo głębokie zrozumienie wewnętrznego działania aplikacji poprzez odczytanie dużej części kodu. Oczywistym miejscem do rozpoczęcia podejścia oddolnego jest główna funkcja, odczytywanie stamtąd w celu zrozumienia programu od jego punktu wejścia do punktu wyjścia. Chociaż ta metoda jest czasochłonna, możesz w pełni zrozumieć program, który pozwala łatwiej odkryć bardziej subtelne błędy.

Selektywne podejście

Oba poprzednie podejścia mają problemy, które uniemożliwiają im skuteczną i terminową metodę lokalizowania błędów. Jednak użycie kombinacji tych dwóch może być bardziej skuteczne. W większości przypadków znaczna część dowolnej bazy kodu to martwy kod, jeśli chodzi o wyszukiwanie luk w zabezpieczeniach. Na przykład przepełnienie bufora w kodzie do analizowania pliku konfiguracyjnego będącego własnością administratora dla serwera WWW nie jest prawdziwym problemem bezpieczeństwa. Aby zaoszczędzić czas i wysiłek, bardziej efektywne jest skoncentrowanie audytu na większości sekcji kodu może zawierać problemy z bezpieczeństwem, które można wykorzystać w rzeczywistych sytuacjach. W selektywnym podejściu do audytu kodu źródłowego audytor zlokalizuje kod, do którego można uzyskać dostęp za pomocą danych wejściowych zdefiniowanych przez atakującego, i skoncentruje rozległą energię audytu na tej części kodu. Przydatne jest jednak dogłębne zrozumienie tego, co robią te krytyczne fragmenty kodu. Jeśli nie wiesz, co robi audytowany fragment kodu lub gdzie pasuje do aplikacji, powinieneś najpierw poświęcić trochę czasu na jego naukę, aby nie tracić czasu na nieopłacalny audyt. Nie ma nic bardziej frustrującego niż znalezienie świetnego błędu w martwym kodzie lub w miejscu, w którym nie można kontrolować danych wejściowych. Ogólnie rzecz biorąc, najbardziej udani audytorzy stosują podejście selektywne. Selektywna metodologia audytu kodu źródłowego jest zazwyczaj najskuteczniejszą ręczną metodą lokalizowania prawdziwych luk w aplikacji.

Klasy podatności

Lista klas błędów, które są często lub nie tak często spotykane w aplikacjach, jest zawsze cenna. Chociaż nasza lista zdecydowanie nie jest kompletna, próbuje ona wypisać większość klas błędów występujących w dzisiejszych aplikacjach. Co kilka lat odkrywana jest nowa klasa błędów i niemal natychmiast odkrywane są nowe luki. Pozostałe luki są wykrywane w miarę upływu czasu; kluczem do ich odnalezienia jest uznanie.

Ogólne błędy logiczne

Chociaż ogólna klasa błędów logicznych luk jest najbardziej niespecyficzna, często jest główną przyczyną wielu problemów. Musisz dość dobrze zrozumieć aplikację, aby znaleźć luki w logice programowania, które mogą prowadzić do warunków bezpieczeństwa. Pomocne jest zrozumienie

wewnętrznych struktur i klas specyficznych dla aplikacji oraz przeprowadzenie burzy mózgów, w których mogą one być niewłaściwie wykorzystywane. Na przykład, jeśli aplikacja używa wspólnej struktury bufora lub klasy ciągu, dogłębne zrozumienie pozwoli Ci wyszukać w aplikacji miejsca, w których elementy struktury lub klasy są niewłaściwie używane lub używane w podejrzany sposób. Podczas audytu w miarę bezpiecznej i dobrze kontrolowanej aplikacji polowanie na ogólne błędy logiczne jest często kolejnym najlepszym sposobem działania.

(Prawie) Wymarłe klasy błędów

Kilka luk w zabezpieczeniach, które były powszechnie znajdowane w oprogramowaniu open source pięć lat temu, zostało teraz wytropionych i prawie wymarło. Tego typu luki generalnie przybierały postać dobrze znanych funkcji nieograniczonego kopiowania pamięci, takich jak strcpy, sprintf i strcat. Chociaż te funkcje mogą być i nadal są bezpiecznie używane w wielu aplikacjach, w przeszłości często zdarzało się, że były one niewłaściwie używane, co mogło prowadzić do przepełnienia bufora. Ogólnie, jednak tego typu luki w zabezpieczeniach nie są już wykrywane w nowoczesnym oprogramowaniu typu open source. Strcpy, sprintf, strcat, gets i wiele podobnych funkcji nie ma pojęcia o rozmiarze swoich buforów docelowych. Pod warunkiem, że bufory docelowe zostały odpowiednio przydzielone lub weryfikacja rozmiaru wejściowego została przeprowadzona przed jakimkolwiek kopiowaniem, możliwe jest korzystanie z większości tych funkcji bez niebezpieczeństwa. Jeśli jednak nie zostanie przeprowadzone odpowiednie sprawdzenie, funkcje te stanowią zagrożenie dla bezpieczeństwa. Mamy teraz znaczną świadomość problemów bezpieczeństwa związanych z tymi funkcjami. Na przykład strony podręcznika sprintf i strcpy wspominają o zagrożeniach związanych z niewykonaniem sprawdzania granic przed wywołaniem tych funkcji. Przykładem tego typu luki jest przepełnienie na serwerze IMAP Uniwersytetu Waszyngtońskiego, naprawiony w 1998 roku. Luka dotyczyła polecenia uwierzytelnienia i była po prostu nieograniczoną kopią ciągu do bufora stosu lokalizowania.

```
char tmp[MAILTMPLEN];

AUTHENTICATOR *auth;

/* make upper case copy of mechanism name */

ucase (strcpy (tmp,mechanism));
```

Konwersja ciągu wejściowego na wielkie litery stanowiła wówczas interesujące wyzwanie dla twórców exploitów; jednak nie stanowi dziś żadnej prawdziwej przeszkody. Poprawka dla luki polegała po prostu na sprawdzeniu rozmiaru ciągu wejściowego i odrzuceniu wszystkiego, co jest zbyt długie.

```
/* cretins still haven't given up */

if (strlen (mechanism) >= MAILTMPLEN)

syslog (LOG_ALERT|LOG_AUTH,"System break-in attempt, host=%.80s",

tcp_clienthost ());
```

Formatowanie ciągów

Klasa luk w formacie łańcucha znaków pojawiła się w 2000 roku i zaowocowała odkryciem istotnych luk w ciągu ostatnich kilku lat. Klasa błędów formatu ciągu jest oparta na tym, że atakujący jest w stanie kontrolować ciąg formatu przekazywany do dowolnej z kilku funkcji, które akceptują printfstyle argumenty (w tym *printf, syslog i podobne funkcje). Jeśli włączone atakujący może kontrolować ciąg formatu, może przekazywać dyrektywy, które spowodują uszkodzenie pamięci i wykonanie dowolnego kodu. Wykorzystywanie tych luk w dużej mierze opiera się na wykorzystaniu wcześniej niejasnej

dyrektywy %n, w której liczba wydrukowanych już bajtów jest zapisana jako argument wskaźnika całkowitego. Ciągi formatujące są bardzo łatwe do znalezienia w audycie. Tylko ograniczona liczba funkcji akceptuje argumenty w stylu printf; dość często wystarczy zidentyfikować wszystkie wywołania tych funkcji i zweryfikować, czy atakujący może kontrolować ciąg formatu.Na przykład poniższe podatne i niepodatne na luki wywołania syslog wyglądają uderzająco inaczej.

Prawdopodobnie zagrożony

syslog(LOG_ERR,ciąg);

Niewrażliwy

syslog(LOG_ERR",%s",ciąg);

Przykładem „prawdopodobnie podatna na zagrożenia" może być zagrożenie bezpieczeństwa, jeśli ciąg znaków jest kontrolowany przez atakującego. Często trzeba prześledzić przepływ danych przez kilka funkcji, aby sprawdzić, czy rzeczywiście istnieje luka w ciągach formatu. Niektóre aplikacje implementują własne, niestandardowe implementacje funkcji typu printflike, a kontrola nie powinna ograniczać się tylko do małego zestawu funkcji. Inspekcja pod kątem błędów w ciągach formatowych jest dość okrojona i wysuszona, i możliwe jest automatyczne określenie, czy błąd istnieje. Najczęstszą lokalizacją, w której można znaleźć błędy ciągu formatu, jest rejestrowanie kodu. Często zdarza się, że ciąg formatu stałego przekazywany jest do funkcji rejestrującej tylko po to, aby dane wejściowe były drukowane do bufora i przekazywane do syslog w sposób podatny na ataki. Poniższy hipotetyczny przykład ilustruje usterkę klasycznego ciągu formatu w kodzie rejestrowania:

```
void log_fn(const char *fmt,...) {

va_list argumenty;

char log_buf[1024];

va_start(argumenty,fmt);

vsnprintf(log_buf,sizeof(log_buf),fmt,args);

va_end(argumenty);

syslog(LOG_NOTICE,log_buf);
```

Luki w ciągach formatujących zostały początkowo ujawnione na serwerze wu-ftpd, a następnie znalezione w wielu innych aplikacjach. Jednakże, ponieważ te luki są bardzo łatwe do znalezienia podczas audytu, zostały prawie wyeliminowane z większości głównych pakietów oprogramowania open source.

Ogólne nieprawidłowe sprawdzanie granic

W wielu przypadkach aplikacje podejmą próbę sprawdzenia granic; jednak dość często zdarza się, że te próby są wykonywane nieprawidłowo. Nieprawidłowe sprawdzanie granic można odróżnić od klas luk, w których nie próbuje się sprawdzać granic, ale w końcu wynik jest taki sam. Oba te typy błędów można przypisać błędom logicznym podczas sprawdzania granic. Chyba że dogłębna analiza prób sprawdzenia granic jest wykonane, te luki mogą nie zostać wykryte. Innymi słowy, nie zakładaj, że fragment kodu nie jest podatny na ataki tylko dlatego, że podejmuje próbę sprawdzenia granic. Sprawdź, czy te próby zostały wykonane poprawnie, zanim przejdziesz dalej. Błąd preprocesora Snort RPC znaleziony przez ISS X-Force na początku 2003 roku jest dobrym przykładem nieprawidłowego sprawdzania granic. W podatnych na ataki wersjach Snort znajduje się następujący kod:

```
while(index < end)
{
/* get the fragment length (31 bits) and move the pointer to the
start of the actual data */
hdrptr = (int *) index;
length = (int)(*hdrptr & 0x7FFFFFFF);
if(length > size)
{
DebugMessage(DEBUG_FLOW, “WARNING: rpc_decode calculated bad
“
“length: %d\n”, length);
return;
}
else
{
total_len += length;
index += 4;
for (i=0; i < length; i++,rpc++,index++,hdrptr++)
*rpc = *index;
}
}
```

W kontekście tej aplikacji długość to długość pojedynczego fragmentu RPC, a rozmiar to rozmiar całego pakietu danych. Bufor wyjściowy jest taki sam jak bufor wejściowy i w dwóch różnych lokalizacjach odwołują się do niego zmienne rpc i index. Kod próbuje ponownie złożyć fragmenty RPC, usuwając nagłówki ze strumienia danych. W każdej iteracji pętli pozycja RPC i index jest zwiększana, a total_len reprezentuje rozmiar danych zapisanych w buforze. Podjęto próbę sprawdzenia granic; jednak ta kontrola jest błędna. Długość bieżącego fragmentu RPC jest porównywana z całkowitym rozmiarem danych. Jednak prawidłowe sprawdzenie polegałoby na porównaniu całkowitej długości wszystkich fragmentów RPC, w tym bieżącego, z rozmiarem bufora. Kontrola jest niewystarczająca, ale nadal występuje, a jeśli dokonasz pobieżnego zbadania kodu, możesz po prostu odrzucić kontrolę jako ważną - ten przykład podkreśla potrzebę weryfikacji sprawdzanie wszystkich granic w ważnych obszarach.

Konstrukcje pętli

Pętle są bardzo powszechnym miejscem znajdowania luk w zabezpieczeniach związanych z przepełnieniem bufora, prawdopodobnie dlatego, że ich zachowanie jest nieco bardziej skomplikowane, z perspektywy programowania, niż kod liniowy. Im bardziej złożona pętla, tym większe

prawdopodobieństwo, że błąd w kodowaniu wprowadzi lukę. Wiele szeroko wdrożonych i krytycznych dla bezpieczeństwa aplikacji zawiera bardzo skomplikowane pętle, z których niektóre są niezabezpieczone. Często programy zawierają pętle w pętlach, co prowadzi do złożonego zestawu interakcji, który jest podatny na błędy. Pętle analizowania lub dowolne pętle przetwarzające dane wejściowe zdefiniowane przez użytkownika są dobrym miejscem do rozpoczęcia inspekcji dowolnej aplikacji, a skupienie się na tych obszarach może dać dobre wyniki przy minimalnym nakładzie pracy. Dobrym przykładem błędnej pętli złożonej jest usterka znaleziona w funkcji crackaddr w Sendmailu przez Marka Dowda. Pętla w tej konkretnej funkcji jest zbyt duża, aby ją tutaj pokazać i znajduje się tam na liście złożonych pętli występujących w oprogramowaniu open source. Ze względu na swoją złożoność i liczba zmiennych manipulowanych w pętli, stan przepełnienia bufora występuje, gdy kilka wzorców danych wejściowych jest przetwarzanych przez tę pętlę. Chociaż w Sendmailu wprowadzono wiele testów, aby zapobiec przepełnieniu bufora, kod nadal miał nieoczekiwane konsekwencje. Kilka niezależnych analiz tej luki, w tym jedna przeprowadzona przez polską grupę badaczy bezpieczeństwa, The Last Stage of Delirium, zaniżyła możliwość wykorzystania tego błędu tylko dlatego, że pominęła jeden z wzorców wejściowych, który doprowadził do przepełnienia bufora.

Luki off-by-one

Luki typu off-by-one lub typu off-by-a-kilka to powszechne błędy kodowania, w których jeden lub bardzo ograniczona liczba bajtów jest zapisywana poza granicami przydzielonej pamięci. Te luki są często wynikiem nieprawidłowego zakończenia ciągów znaków null i są często znajdowane w pętlach lub wprowadzane przez typowe funkcje ciągów. W wielu przypadkach te luki nadająsię do wykorzystania i zostały znalezione w niektórych szeroko wdrożonych aplikacjach w przeszłości. Na przykład następujący kod został znaleziony w Apache 2 przed wersją 2.0.46 i został nieco po cichu załatany:

```
if (last_len + len > alloc_len) {

char *fold_buf;

alloc_len += alloc_len;

if (last_len + len > alloc_len) {

alloc_len = last_len + len;

}

fold_buf = (char *)apr_palloc(r->pool, alloc_len);

memcpy(fold_buf, last_field, last_len);

last_field = fold_buf;

}

memcpy(last_field + last_len, field, len +1); /* +1 for nul */
```

W tym kodzie, który dotyczy nagłówków MIME wysyłanych jako część żądania do serwera WWW, jeśli pierwsze dwie instrukcje if są prawdziwe, przydzielony bufor będzie o 1 bajt za mały. Ostatnie wywołanie memcpy zapisze bajt null poza zakresem. Wykorzystanie tego błędu okazało się bardzo trudne ze względu na niestandardową implementację sterty; jest to jednak rażący przypadek, w którym pojedyncza osoba ma do czynienia z zakończeniem zerowym. Każda pętla, która kończy ciąg znaków

null na końcu, powinna być dwukrotnie sprawdzona pod kątem warunku wyłączenia o jeden. Poniższy kod, znaleziony wewnątrz demona ftp OpenBSD, ilustruje problem:

```
char npath[MAXPATHLEN];

int i;

for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++)

{

npath[i] = *name;

if (*name == '"')

npath[++i] = '"';

}

npath[i] = '\0';
```

Chociaż kod próbuje zarezerwować miejsce na bajt o wartości null, jeśli ostatni znak na granicy bufora wyjściowego jest cudzysłowem, występuje warunek wykluczenia o jeden. Niektóre funkcje biblioteczne wprowadzają warunki o jeden po drugim, jeśli są używane niewłaściwie. Na przykład funkcja strncat zawsze kończy null-kończy swój łańcuch wyjściowy i zapisuje bajt null poza zakresem, jeśli nie zostanie poprawnie wywołana z trzecim argumentem równym ilości miejsca pozostałego w buforze wyjściowym pomniejszonym o jeden dla bajtu null. Poniższy przykład pokazuje nieprawidłowe użycie strncat:

```
strcpy(buf,"Test:");

strncat(buf,input,sizeof(buf)-strlen(buf));
```

Bezpieczne użycie:

```
strncat(buf,input,sizeof(buf)-strlen(buf)-1);
```

Problemy z niezerowym zakończeniem

Aby ciągi były obsługiwane w sposób bezpieczny, muszą być właściwie zakończone znakiem null, aby można było łatwo określić ich granice. Nieprawidłowo zakończone łańcuchy mogą prowadzić do możliwego do wykorzystania problemu z bezpieczeństwem w późniejszym czasie wykonywania programu. Na przykład, jeśli ciąg nie jest poprawnie zakończony, sąsiednia pamięć może być uważana za część tego samego ciągu. Ta sytuacja może mieć kilka skutków, takich jak znaczne zwiększenie długości ciągu lub spowodowanie, że operacje modyfikujące ciąg uszkadzają pamięć poza granicami bufora ciągu. Niektóre funkcje biblioteczne z natury wprowadzają problemy związane z zakończeniem zerowym i należy ich szukać podczas audytu kodu źródłowego. Na przykład, jeśli funkcja strncpy zabraknie miejsca w buforze docelowym, nie zakończy ona zerem napisanego przez siebie ciągu. Programista musi jawnie wykonać zakończenie zerowe lub zaryzykować potencjalną lukę w zabezpieczeniach. Na przykład poniższy kod nie byłby bezpieczny:

```
char dest_buf[256];

char not_term_buf[256];

strncpy(not_term_buf,input,sizeof(non_term_buf));
```

strcpy(dest_buf,not_term_buf);

Ponieważ pierwszy strncpy nie zakończy wartości null non_term_buf, drugi strcpy nie jest bezpieczny, mimo że oba bufory mają ten sam rozmiar. Poniższy wiersz, wstawiony pomiędzy strncpy i strcpy, zabezpieczy kod przed przepełnieniem bufora:

not_term_buf[sizeof(not_term_buf) – 1] = 0;

Możliwość wykorzystania tych problemów jest nieco ograniczona przez stan sąsiednich buforów, ale w wielu przypadkach błędy te mogą prowadzić do wykonania dowolnego kodu.

Pomijanie problemów z zakończeniem zerowym

Niektóre możliwe do wykorzystania błędy kodowania w aplikacjach wynikają z możliwości pominięcia bajtu kończącego znak zerowy w ciągu i kontynuowania przetwarzania w niezdefiniowanych regionach pamięci. Po pominięciu bajtu kończącego znak null, jeśli jakiekolwiek dalsze przetwarzanie spowoduje operację zapisu, może być możliwe spowodowanie uszkodzenia pamięci, które prowadzi do wykonania dowolnego kodu. Te luki zwykle pojawiają się w pętlach przetwarzania ciągów, zwłaszcza gdy przetwarzany jest więcej niż jeden znak na raz lub gdy założenia dotyczące długości ciągu są wykonane. Poniższy przykład kodu był do niedawna obecny w module mod_rewrite w Apache:

```
else if (is_absolute_uri(r->filename)) {

/* it was finally rewritten to a remote URL */

/* skip 'scheme:' */

for (cp = r->filename; *cp != ':' && *cp != '\0'; cp++)

;

/* skip '://' */

cp += 3;
```

where is_absolute_uri does the following:

```
int i = strlen(uri);

if ( (i > 7 && strncasecmp(uri, "http://", 7) == 0)

|| (i > 8 && strncasecmp(uri, "https://", 8) == 0)

|| (i > 9 && strncasecmp(uri, "gopher://", 9) == 0)

|| (i > 6 && strncasecmp(uri, "ftp://", 6) == 0)

|| (i > 5 && strncasecmp(uri, "ldap:", 5) == 0)

|| (i > 5 && strncasecmp(uri, "news:", 5) == 0)

|| (i > 7 && strncasecmp(uri, "mailto:", 7) == 0) ) {

return 1;

}

else {
```

return 0;

}

Problemem jest tutaj linia c += 3; w którym kod przetwarzania próbuje pominąć :// w identyfikatorze URI. Zauważ jednak, że w obrębie is_absolute_uri nie wszystkie schematy URI kończą się na ://. Gdyby zażądano identyfikatora URI, który był po prostu ldap:a, bajt kończący znak null zostałby pominięty przez kod. Przechodząc dalej do przetwarzania tego identyfikatora URI, zapisywany jest w nim bajt null, co sprawia, że luka ta może być potencjalnie wykorzystana. W tym konkretnym przypadku, aby to zadziałało, muszą istnieć pewne reguły przepisywania, ale takie problemy są nadal dość powszechne w wielu bazach kodu open source i powinny być brane pod uwagę podczas audytu.

Luki ze znakiem w porównaniach

Wielu programistów próbuje sprawdzać długość danych wprowadzanych przez użytkownika, ale często jest to wykonywane niepoprawnie, gdy używane są specyfikatory długości ze znakiem. Wiele specyfikatorów długości, takich jak size_t, nie ma znaku i nie jest podatnych na te same problemy, co specyfikatory długości ze znakiem, takie jak off_t. Jeśli porównuje się dwie liczby całkowite ze znakiem, kontrola długości może nie uwzględniać możliwości, że liczba całkowita jest mniejsza od zera, zwłaszcza w porównaniu z wartością stałą. Standardy porównywania liczb całkowitych różnych typów niekoniecznie wynikają z zachowania skompilowanego kodu i musimy podziękować przyjacielowi za wskazanie właściwego zachowania kompilatora. Zgodnie ze standardem ISO C, jeśli porównuje się dwie liczby całkowite różnych typów lub rozmiarów, są one najpierw konwertowane na typ ze znakiem int, a następnie porównywane. Jeśli którakolwiek z liczb całkowitych jest większa niż rozmiar int ze znakiem, obie są konwertowane na większy typ, a następnie porównywane. Typ bez znaku jest większy niż typ ze znakiem i będzie miał pierwszeństwo podczas porównywania int bez znaku i int ze znakiem. Na przykład następujące porównanie byłoby bez znaku:

if((int)left < (unsigned int)right)

Jednak to porównanie byłoby ze znakiem:

if((int)left < 256)

Niektóre operatory, takie jak operator sizeof(), są bez znaku. Poniższe porównanie byłoby bez znaku, mimo że wynik operatora sizeof jest wartością stałą:

if((int) left < sizeof(buf))

Poniższe porównanie byłoby jednak ze znakiem, ponieważ obie krótkie liczby całkowite są konwertowane na liczbę całkowitą ze znakiem przed porównaniem:

if((unsigned short)a < (short)b)

W większości przypadków, zwłaszcza gdy używane są 32-bitowe liczby całkowite, musisz mieć możliwość bezpośredniego określenia liczby całkowitej, aby ominąć te sprawdzenia rozmiaru. Na przykład w praktycznym przypadku nie będzie możliwe spowodowanie, aby strlen() zwróciła wartość, którą można rzutować na ujemną liczbę całkowitą, ale jeśli liczba całkowita zostanie w jakiś sposób pobrana bezpośrednio z pakietu, często będzie to możliwe żeby to było negatywne. Podpisana luka porównania doprowadziła do luki Apache z kodowaniem fragmentarycznym, odkrytej w 2002 roku przez Marka Litchfielda z NGSSoftware. Ten fragment kodu był winowajcą:

len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;

len_read = ap_bread(r->connection->client, buffer, len_to_read);

W tym przypadku bufsiz jest liczbą całkowitą ze znakiem, określającą ilość wolnego miejsca w buforze, a r->remaining jest typem ze znakiem off_t, określającym rozmiar porcji bezpośrednio z żądania. Zmienna len_to_read ma być minimalną wartością bufsiz lub r->remaining, ale jeśli rozmiar porcji jest wartością ujemną, można pominąć to sprawdzenie. Kiedy ujemna wielkość porcji jest przekazywana do ap_bread, jest rzutowana na bardzo dużą wartość dodatnią, co skutkuje bardzo dużą wartością memcpy. Ten błąd był oczywiście i łatwy do wykorzystania w Win32 poprzez nadpisanie SEH, a Gobbles Security Group sprytnie udowodniła, że można go wykorzystać w BSD z powodu błędu w ich implementacji memcpy. Tego typu luki w zabezpieczeniach są nadal obecne w oprogramowaniu i warto je sprawdzić za każdym razem, gdy podpisano liczby całkowite używane jako specyfikatory długości.

Luki związane z liczbami całkowitymi

Przepełnienia liczb całkowitych wydają się być modnym słowem, którego badacze bezpieczeństwa używają do opisywania wielu luk w zabezpieczeniach, z których wiele nie jest w rzeczywistości związanych z przepełnieniami liczb całkowitych. Przepełnienia liczb całkowitych zostały po raz pierwszy dobrze zdefiniowane w przemówieniu „Professional Source Code Auditing" wygłoszonym na BlackHat USA 2002, chociaż te przepełnienia były znane i identyfikowane przez badaczy bezpieczeństwa już od jakiegoś czasu. Przepełnienia liczby całkowitej występują, gdy liczba całkowita wzrasta powyżej swojej wartości maksymalnej lub spada poniżej wartości minimalnej. Maksymalna lub minimalna wartość liczby całkowitej jest określona przez jej typ i rozmiar. 16-bitowa liczba całkowita ze znakiem ma maksymalną wartość 32 767 (0x7fff) i minimalną wartość –32 768 (-0x8000). 32-bitowa liczba całkowita bez znaku ma maksymalną wartość 4 294 967 295 (0xffffffff) i minimalną wartość 0. Jeśli 16-bitowa liczba całkowita ze znakiem, która ma wartość 32 767 zostanie zwiększona o jeden, jej wartość stanie się –32 768 w wyniku całkowitą przepełnienie. Przepełnienia liczb całkowitych są przydatne, gdy chcesz pominąć sprawdzanie rozmiaru lub spowodować alokację buforów o rozmiarze zbyt małym, aby pomieścić skopiowane do nich dane. Przepełnienia liczb całkowitych można ogólnie podzielić na jedną z dwóch kategorii: przepełnienia dodawania i odejmowania lub przepełnienia mnożenia. Dodawanie lub odejmowanie przepełnienia wynikają, gdy dwie wartości są dodawane lub odejmowane, a operacja powoduje zawinięcie wyniku poza granicę maksymalną/minimalną dla typu liczb całkowitych. Na przykład poniższy kod może spowodować potencjalne przepełnienie liczby całkowitej:

```
char *buf;

int allocation_size = attacker_defined_size + 16;

buf = malloc(allocation_size);
```

W tym przypadku, jeśli attacker_defined_size ma wartość pomiędzy –16 a –1, dodanie spowoduje przepełnienie liczby całkowitej, a wywołanie malloc() przydzieli bufor o wiele za mały, aby pomieścić dane skopiowane w wywołaniu memcpy(). Taki kod jest bardzo powszechny w aplikacjach typu open source. Z wykorzystaniem tych podatności wiążą się trudności, ale mimo to te błędy istnieją. Przepełnienia odejmowania można często znaleźć, gdy program oczekuje, że dane wejściowe użytkownika będą miały minimalną długość. Poniższy kod byłby podatny na przepełnienie liczby całkowitej:

```
#define HEADER_SIZE 16

char data[1024],*dest;
```

```
int n;

n = read(sock,data,sizeof(data));

dest = malloc(n);

memcpy(dest,data+HEADER_SIZE,n – HEADER_SIZE);
```

W tym przykładzie zawinięcie liczby całkowitej występuje w trzecim argumencie do memcpy, jeśli dane odczytane z sieci są mniejsze niż oczekiwany minimalny rozmiar (HEADER_SIZE). Przepełnienia mnożenia występują, gdy dwie wartości są mnożone przez siebie, a wynikowa wartość przekracza maksymalny rozmiar dla typu liczb całkowitych. Ten typ luki został znaleziony w OpenSSH oraz w bibliotece RPC firmy Sun w 2002 roku. Poniższy kod z OpenSSH (przed 3.4) jest klasycznym przykładem przepełnienia mnożenia:

```
nresp = packet_get_int();

if (nresp > 0) {

response = xmalloc(nresp * sizeof(char*));

for (i = 0; i < nresp; i++)

response[i] = packet_get_string(NULL);

}
```

W tym przypadku nresp jest liczbą całkowitą bezpośrednio z pakietu SSH. Jest mnożony przez rozmiar wskaźnika znakowego, w tym przypadku 4, i ten rozmiar jest przydzielany jako bufor docelowy. Jeśli nresp zawiera wartość większą niż 0x3fffffff, to mnożenie przekroczy maksymalną wartość dla liczby całkowitej bez znaku i przepełnienia. Możliwe jest spowodowanie bardzo małej alokacji pamięci i skopiowanie do niej dużej liczby wskaźników znakowych. Co ciekawe, ta konkretna luka została wykorzystana w OpenBSD ze względu na bezpieczniejszą implementację sterty w OpenBSD, która nie przechowuje struktury kontrolnej w linii na stercie. W przypadku implementacji sterty z wbudowanymi strukturami kontrolnymi, hurtowe uszkodzenie dowolnej sterty ze wskaźnikami doprowadziłoby do awarii kolejnych alokacji, takich jak ta w ciągu package_get_string. Liczby o mniejszych rozmiarach są bardziej podatne na przepełnienia; możliwe jest wywołanie zawijania liczb całkowitych dla 16-bitowych typów liczb całkowitych za pomocą popularnych procedur, takich jak strlen(). Ten typ przepełnienia liczb całkowitych był odpowiedzialny za przepełnienie w RtlDosPathNameToNtPathName_U, które doprowadziło do powstania luki w zabezpieczeniach IIS WebDAV opisanej w biuletynie Microsoft Security Bulletin MS03-007. Luki związane z liczbami całkowitymi są bardzo istotne i nadal są dość powszechne. Chociaż wielu programistów zdaje sobie sprawę z niebezpieczeństw związanych z operacjami związanymi z ciągami znaków, zwykle są mniej świadomi niebezpieczeństw związanych z manipulowaniem liczbami całkowitymi. Podobne luki prawdopodobnie będą pojawiać się w programach jeszcze przez wiele lat.

Konwersje liczb całkowitych różnej wielkości

Konwersje między liczbami całkowitymi o różnych rozmiarach mogą mieć interesujące i nieoczekiwane wyniki. Te konwersje mogą być niebezpieczne, jeśli ich konsekwencje nie zostaną dokładnie rozważone, a jeśli zostaną zauważone w kodzie źródłowym, należy je dokładnie zbadać. Mogą prowadzić do obcięcia wartości, spowodować zmianę znaku liczby całkowitej lub rozszerzenie wartości, a czasami mogą prowadzić do warunków bezpieczeństwa, które można wykorzystać. Konwersja z typu dużej liczby całkowitej do typu mniejszej liczby całkowitej (od 32 do 16 bitów lub od 16 do 8 bitów)

może skutkować obcięciem wartości lub przełączeniem znaku. Na przykład, jeśli 32-bitowa liczba całkowita ze znakiem i ujemna wartość –65 535 zostanie zmieniona na 16-bitową liczbę całkowitą, wynikowa 16-bitowa liczba całkowita będzie miała wartość +1 z powodu obcięcia najwyższych 16 bitów liczba całkowita. Konwersje z mniejszych na większe typy liczb całkowitych mogą spowodować rozszerzenie znaku, w zależności od typu źródłowego i docelowego. Na przykład konwersja 16-bitowej liczby całkowitej ze znakiem o wartości -1 na 32-bitową liczbę całkowitą bez znaku da w wyniku wartość o 4 GB mniejszą o jeden. Wykres w tabeli 1 może pomóc w śledzeniu konwersji jednego rozmiaru liczb całkowitych na inny. Jest to właściwe dla ostatnich wersji GCC

ŹRÓDŁO ROZMIAR / ŹRÓDŁO PRZEZNACZENIE PRZEZNACZENIE TYP WARTOŚĆ ROZMIAR/TYP WARTOŚĆ

16-bitowy ze znakiem –1 (0xffff) 32-bitowy bez znaku 4294967295 (0xffffffff)

16-bitowy ze znakiem –1 (0xffff) 32-bitowy ze znakiem –1 (0xffffffff)

16-bitowy bez znaku 65535 (0xffff) 32-bitowy bez znaku 65535 (0xffff)

16-bitowy bez znaku 65535 (0xffff) 32-bitowy ze znakiem 65535 (0xffff)

32-bitowy ze znakiem -1 (0xffffffff) 16-bitowy bez znaku 65535 (0xffff)

32-bitowy ze znakiem –1 (0xffffffff) 16-bitowy ze znakiem –1 (0xffffff)

32-bitowy bez znaku 32768 (0x8000) 16-bitowy bez znaku 32768 (0x8000)

32-bitowy bez znaku 32768 (0x8000) 16-bitowy ze znakiem –32768 (0x8000)

32-bitowy ze znakiem –40960 (0xffff6000) 16-bitowy ze znakiem 24576 (0x6000)

Mamy nadzieję, że ta tabela pomoże wyjaśnić wzajemne przeliczanie liczb całkowitych o różnych rozmiarach. Niedawna luka wykryta w Sendmailu w ramach funkcji wstępnego skanowania jest dobrym przykładem tego typu luki. Znak ze znakiem (8 bitów) został pobrany z bufora wejściowego i przekonwertowany na 32-bitową liczbę całkowitą ze znakiem. Znak ten został rozszerzony do 32-bitowej wartości -1, która jest również definicją sytuacji specjalnej NOCHAR. Prowadzi to do niepowodzenia sprawdzania granic tej funkcji i możliwego do wykorzystania zdalnego przepełnienia bufora. Wzajemna konwersja liczb całkowitych o różnych rozmiarach jest wprawdzie dość skomplikowana i może być źródłem błędów, jeśli nie jest dobrze przemyślana w aplikacjach. Istnieje kilka prawdziwych powodów, dla których w nowoczesnych aplikacjach używa się liczb całkowitych o różnej wielkości; jeśli zauważysz je podczas audytu, warto poświęcić czas na dogłębne zbadanie ich wykorzystania.

Podwójne wolne luki

Chociaż błąd polegający na dwukrotnym zwolnieniu tej samej porcji pamięci może wydawać się całkiem niegroźny, może prowadzić do uszkodzenia pamięci i wykonania dowolnego kodu. Niektóre implementacje sterty są odporne lub odporne na tego typu wady, a ich wykorzystanie jest ograniczone do niektórych platform. Większość programistów nie popełnia błędu polegającego na dwukrotnym zwolnieniu zmiennej lokalnej (chociaż widzieliśmy to). Podwójnie wolne luki są najczęściej znajdowane, gdy bufory sterty są przechowywane we wskaźnikach o zasięgu globalnym. Wiele aplikacji po zwolnieniu wskaźnika globalnego ustawi wskaźnik na wartość null, aby zapobiec jego ponownemu użyciu. Jeśli aplikacja nie robi czegoś podobnego, dobrym pomysłem jest rozpoczęcie polowania na miejsca, w których fragment pamięci można zwolnić dwukrotnie. Tego typu luki mogą również wystąpić w kodzie C++, gdy niszczysz instancję klasy, z której niektórzy członkowie zostali już zwolnieni.

Niedawna luka w zlib została odkryta, w której zmienna globalna była zwalniana dwukrotnie, gdy podczas dekompresji wystąpił pewien błąd. Ponadto niedawna luka w serwerze CVS była również wynikiem podwójnego zwolnienia.

Luki dotyczące wykorzystania pamięci poza zakresem

Niektóre regiony pamięci w aplikacji mają zakres i okres istnienia, dla których są ważne. Każde użycie tych regionów, zanim staną się ważne lub gdy staną się nieważne, można uznać za zagrożenie bezpieczeństwa. Potencjalnym skutkiem jest uszkodzenie pamięci, które może prowadzić do wykonania dowolnego kodu.

Niezainicjowane użycie zmiennej

Chociaż stosunkowo rzadko można zobaczyć użycie niezainicjowanych zmiennych, te luki pojawiają się raz na jakiś czas i mogą prowadzić do rzeczywistych warunków do wykorzystania w aplikacjach. Pamięć statyczna, taka jak ta w sekcji .data lub .bss pliku wykonywalnego, jest inicjowana do wartości null podczas uruchamiania programu. Nie masz takiej gwarancji dla zmiennych stosu lub sterty i muszą one zostać jawnie zainicjowane przed odczytem, aby zapewnić spójne wykonywanie programu.

Jeśli zmienna jest niezainicjowana, jej zawartość jest z definicji niezdefiniowana. Jednak możliwe jest dokładne przewidzenie, jakie dane będzie zawierał niezainicjowany obszar pamięci. Na przykład lokalna zmienna stosu, której nie zainicjował, będzie zawierać dane z poprzednich wywołań funkcji. Może zawierać dane argumentów, zapisane rejestry lub zmienne lokalne z poprzednich wywołań funkcji, w zależności od ich położenia na stosie. Jeśli atakujący ma szczęście kontrolować odpowiednią część pamięci, często może wykorzystać tego typu luki. Niezainicjowane luki w zabezpieczeniach zmiennych są rzadkie, ponieważ mogą prowadzić do natychmiastowych awarii programu. Najczęściej można je znaleźć w kodzie, który nie jest często wykonywany, na przykład bloki kodu rzadko wyzwalane z powodu nietypowych warunków błędów. Wiele kompilatorów będzie próbowało wykryć użycie niezainicjowanych zmiennych. Microsoft Visual C++ ma pewną logikę wykrywania tego typu warunków, a gcc również podejmuje dobrą próbę zlokalizowania tych problemów, ale żaden z nich nie jest doskonały; dlatego to na deweloperze spoczywa obowiązek nie popełniania tego rodzaju błędów. Poniższy hipotetyczny przykład pokazuje zbyt uproszczony przypadek użycia niezainicjowanej zmiennej:

```
int vuln_fn(char *data,int some_int) {

char *test;

if(data) {

test = malloc(strlen(data) + 1);

strcpy(test,data);

some_function(test);

}

if(some_int < 0) {

free(test);

return -1;

free(test);
```

return 0;

}

W takim przypadku, jeśli dane argumentu mają wartość NULL, test wskaźnika nie jest inicjowany. Ten wskaźnik byłby wtedy w stanie niezainicjowanym, gdy zostanie zwolniony później w funkcji. Zauważ, że ani gcc, ani Visual C++ nie ostrzegają programisty o błędzie w czasie kompilacji. Chociaż ten rodzaj luki nadaje się do automatycznego wykrywania, niezainicjowane błędy związane ze zmiennym użyciem wciąż znajdują się w dzisiejszych aplikacjach. Chociaż niezainicjowane zmienne luki w zabezpieczeniach są dość rzadkie, są również dość subtelne i mogą pozostać niewykryte przez lata.

Użyj po darmowych lukach

Bufory sterty są ważne przez cały okres życia, od czasu ich alokacji do czasu ich cofnięcia alokacji przez free lub realloc o rozmiarze zero. Wszelkie próby zapisu w buforze sterty po jego zwolnieniu mogą prowadzić do uszkodzenia pamięci i ostatecznie wykonania dowolnego kodu. Użycie po zwolnieniu luk jest najbardziej prawdopodobne, gdy kilka wskaźników do bufora sterty jest przechowywanych w różnych lokalizacjach pamięci, a jeden z nich jest zwolniony, lub gdy używane są wskaźniki do różnych przesunięć w buforze sterty, a oryginalny bufor jest zwolniony. Ten rodzaj luki może powodować niewyjaśnione uszkodzenie sterty i zwykle jest wykorzeniony w procesie rozwoju. Użyj po darmowych lukach, które wkradają się do wydanych wersji oprogramowania, najprawdopodobniej w obszarach kodu, które są rzadko używane lub które radzą sobie z nietypowymi błędami. Luka Apache 2 psprintf ujawniona w maju 2003 roku była przykładem luki use after free, w której aktywny węzeł pamięci został przypadkowo uwolniony, a następnie rozdany przez procedurę alokacji podobną do Malloc Apache.

Problemy wielowątkowe i bezpieczny kod ponownego wejścia

Większość aplikacji open source nie jest wielowątkowa; jednak aplikacje, które są takie, niekoniecznie podejmują środki ostrożności, aby zapewnić, że są bezpieczne wątkowo. Każdy wielowątkowy kod, w którym te same zmienne globalne są dostępne przez różne wątki bez odpowiedniego blokowania, może prowadzić do potencjalnych problemów z bezpieczeństwem. Ogólnie rzecz biorąc, te błędy nie są wykrywane, chyba że aplikacja jest mocno obciążona, i mogą pozostać niewykryte lub zostać odrzucone jako sporadyczne błędy oprogramowania, które nigdy nie są weryfikowane. Jak nakreślił Michał Zalewski w Problemach z Msktemp() , dostarczanie sygnałów w systemie Unix może spowodować zatrzymanie wykonywania, gdy zmienne globalne znajdują się w nieoczekiwanym stanie. Jeśli w procedurach obsługi sygnałów używane są funkcje biblioteczne, które nie są bezpieczne dla ponownego wchodzenia, może to prowadzić do uszkodzenia pamięci. Chociaż istnieją bezpieczne dla wątków i ponownie wchodzące wersje wielu funkcji, nie zawsze są one używane w kodzie wielowątkowym lub podatnym na ponowne wchodzenie. Inspekcja pod kątem tych luk wymaga pamiętania o możliwości wystąpienia wielu wątków. Bardzo pomocne jest zrozumienie, co robią podstawowe funkcje biblioteczne, ponieważ mogą one być źródłem problemów. Jeśli będziesz pamiętać te koncepcje, problemy związane z wątkami nie będą wyjątkowo trudne do zlokalizowania.

Nie do poznania: prawdziwa podatność kontra błąd

Wiele razy można zidentyfikować błąd oprogramowania, który nie stanowi prawdziwej luki w zabezpieczeniach. Badacze bezpieczeństwa muszą zrozumieć zakres i wpływ luki przed podjęciem dalszych kroków. Chociaż często nie jest możliwe potwierdzenie pełnego wpływu błędu, dopóki nie zostanie on pomyślnie wykorzystany, większość bardziej żmudnych prac związanych z bezpieczeństwem można wykonać za pomocą prostej analizy kodu źródłowego. Przydatne jest prześledzenie wstecz od punktu podatności, aby określić, czy można spełnić niezbędne wymagania,

aby wyzwolić podatność. Upewnij się, że luka rzeczywiście znajduje się w aktywnym kodzie i że osoba atakująca może kontrolować wszystkie niezbędne zmienne, a także sprawdź, czy w dalszej części przepływu kodu nie ma żadnych oczywistych kontroli, które mogłyby zapobiec wyzwoleniu błędu. Często należy sprawdzać pliki konfiguracyjne dystrybuowane z oprogramowaniem, aby określić, czy funkcje opcjonalne są często włączane lub wyłączane. Te proste testy mogą zaoszczędzić dużo czasu na opracowywanie exploitów i pomóc uniknąć frustracji związanej z tworzeniem kodu wykorzystującego exploita, który nie dotyczy problemu.

Podsumowanie

Badanie luk w zabezpieczeniach może być czasem frustrującym zadaniem, ale innym razem jest to świetna zabawa. Jako audytor będziesz szukał czegoś, co może w rzeczywistości nie istnieć; musisz mieć wielką determinację, aby znaleźć coś wartościowego. Szczęście może oczywiście pomóc, ale konsekwentne badanie podatności zwykle oznacza godziny żmudnego audytu i dokumentacji. Czas wielokrotnie udowodnił, że każdy większy pakiet oprogramowania zawiera luki w zabezpieczeniach, które można wykorzystać. Ciesz się audytem.