Największa tajemnica CROOK-5

Na przestrzeni niemal trzech lat, które minęły od pierwszego uruchomienia CROOK-5 w emulatorze MERY-400, system operacyjny z Politechniki Gdańskiej stawiał wiele zagadek. Większość z nich wynikała z niedostatków w emulacji komputera. System czasami odmawiał współpracy, a jego narzędzia nie zawsze spełniały swoje funkcje. Sytuacja poprawiła się znacznie, gdy po kolejnych poprawkach emulator przeszedł wreszcie poprawnie testy procesora i arytmometru dostarczane przez producenta MERY-400, a kiedy udokumentowane zostały i zaimplementowane wszystkie przeróbki procesora, CROOK i narzędzia systemowe przestały sprawiać jakiekolwiek problemy.

Niestety nie można było tego samego powiedzieć o programach użytkowych. Niektóre z nich od czasu do czasu kończyły się błędami, inne niby działały, ale konsekwentnie odmawiały spełniania jakiejś wybranej funkcji. Spektrum problemów było szerokie, a częstotliwość i okoliczności ich występowania wydawały się być przypadkowe. Dotyczyły one jednak zawsze grupy tych samych binariów.

Szczególnie problematyczny okazał się kompilator C. Tylko jedna na kilkanaście kompilacji przebiegała poprawnie. Pozostałe kończyły się smutnym *„error in program”*, tragicznym *„WRONG INSTRUCTION”*, dziwnymi *„ERR BRAK O^PERATORA”* lub *„ERR BR^AK ZMIENNEJ”*, czy wreszcie zupełnym zapętleniem się programu.

Powtarzalność problemów była w tym przypadku na tyle wysoka, że kompilator C stał się idealnym kandydatem do podjęcia próby rozwiązania największej do tej pory zagadki CROOK-a.

Winny CROOK?

Gdzie więc leży przyczyna? Za każdym razem, kiedy pod kontrolą systemu CROOK w emulatorze dzieje się coś dziwnego, jedna hipoteza powraca jak mantra: czy z tą wersją systemu aby na pewno działały poprawnie wszystkie programy?

Obraz dysku z systemem uruchamianym w emulatorze EM400 pochodzi z maszyny, która do końca swoich dni była sprawna i wykorzystywana przez autorów CROOK-a do jego rozwoju. Nowe funkcjonalności były dodawane, inne usprawniane. Czy na pewno zachowana była ciągła kompatybilność? Niepokój jest tym bardziej uzasadniony, że dziwną plagą dotknięte były głównie nowsze binaria. Budowane narzędziami, które nie istniały dla starszych wersji systemu, na przykład konsolidatorem LINK stworzonym w Instytucie Informatyki Uniwersytetu Warszawskiego.

Hipotezie winiącej CROOK-a przeczy jednak jeden mocny fakt. W przypadku kompilatora C dziwne zachowania powtarzają się na wszystkich jądrach systemu uczestniczących w testach: 8/15, 8/14, 8/7 i 7/6. Trudno sobie wyobrazić, że na maszynie używanej regularnie przez kilku programistów, kompilator C był niesprawny na przestrzeni wielu miesięcy.

Winny EM400?

Mimo setek dni spędzonych na zgłębianiu szczegółów pracy procesora MERY-400 i implementowaniu ich w emulatorze, wciąż najbardziej prawdopodobną przyczyną pozostaje błąd w emulacji.

EM400 przygotowany jest na badanie takich sytuacji i pozwala z dużą dokładnością śledzić wszystko to, co dzieje się w emulowanym systemie. Może zapisywać do pliku log zawierający nie tylko każdy krok procesora, ale również wywołania systemowe, operacje I/O, obsługę przerwań, etc. Choć taki log ma nieraz kilkaset megabajtów i jego analiza bywa żmudna, to jest to niemal pewna droga do sukcesu.

Zacznijmy więc od takiego właśnie podejrzenia działań kompilatora C. Już kilka minut analizy zapisów w logu pod kątem najbardziej prawdopodobnych anomalii prowadzi do następującego fragmentu, wskazującego bezpośrednią przyczynę problemu:

```
CPU 2 | USR  2:0xf1e9 CC0    |     AWT r7, -1        T = -1
CPU 2 | USR  2:0xf1ea CC0    |     AW r7, [r2]       N = 0x53c0 = 21440
CPU 2 | USR  2:0xf1eb CC0    |     .word 0x0022
CPU 2 | USR  2:0xf1eb CC0    |     (ineffective: illegal instruction)
```

Mówi on, że w trakcie wykonywania procesu w przestrzeni użytkownika, pod adresem *0xf1eb* w bloku pamięci nr *2*, procesor trafił na nielegalną instrukcję o kodzie *0x22*. Dziwne, bo deasemblacja zbioru z programem pokazuje, że powinna w tym miejscu być zupełnie legalna instrukcja:

```
0xf1e9:   AWT   r7, -1
0xf1ea:   AW    r7, [r2]
0xf1eb:   TW    r6, word_5944
```

Podejrzenie o błąd w emulatorze nabiera sensu. Możliwe wyjaśnienia takiej sytuacji dzielą się zasadniczo na dwie grupy:

1. Emulator dokonuje niepoprawnego odczytu pamięci, mimo że w komórce znajduje się poprawna wartość (przyczyną może być na przykład niespójny stan pamięci między wątkami emulacji, czy błąd w emulacji adresowania).
2. Wartość *0x22* została do komórki faktycznie zapisana (np. z powodu błędu w adresowaniu, czy błędu po stronie urządzenia I/O, piszącego do pamięci), a odczyt jest poprawny.

Większość potencjalnych błędów z pierwszej grupy można szybko wyeliminować testami, w które nie ma się co tutaj zagłębiać. Pozostaje w zasadzie tylko druga, znacznie bardziej prawdopodobna możliwość: coś nadpisuje zawartość nieszczęsnej komórki *0xf1eb* w segmencie pamięci kompilatora. Po dodaniu śledzenia zapisów pod ten adres okazuje się jednak, że oprócz wstępnego ładowania obrazu procesu, poszukiwany zapis nie jest w ogóle wykonywany! Przy kolejnych uruchomieniach kompilatora komórka *0xf1eb* zawiera poprawny rozkaz, a kompilator jak nie działał, tak nie działa.

Spróbujmy w takim razie zebrać większą próbkę wystąpień pierwotnie zauważonego problemu. Krok wstecz i kolejne logi z różnych wywołań CC0 pokazują, że nielegalne instrukcje owszem, wciąż pojawiają się w kodzie kompilatora, ale pod różnymi, losowo wyglądającymi (sic!) adresami. W dodatku niemal zawsze mają ten sam kod: *0x22*.

Dobrze, postawmy więc pytanie nieco inaczej: Co (skąd pochodząca instrukcja) zapisuje wartość *0x22* gdzieś (gdziekolwiek) w przestrzeni adresowej procesu użytkownika? Tym razem uzyskanie odpowiedzi wymaga nieco więcej zachodu, ale ostatecznie odnajduje się ona w tym samym logu emulacji w postaci zapisu:

```
CPU 2 |  OS  2:0x137a CC0    |     PW r6, r1        N = 0xf1eb = -3605
```

Ta jedna, niewinnie wyglądająca linia, jest niczym obuch celnie wymierzony w czerep autora niniejszego tekstu. Znaczy bowiem, że pamięć procesu niszczona jest przez jądro systemu CROOK.

Jednak CROOK

Ile dobrej woli by nie włożyć w interpretację tego faktu, to nadpisywanie rozkazów w obszarze procesu użytkownika nie należy do standardowych zadań systemu operacyjnego. Z czym więc mamy do czynienia? Zwykły błąd? Brutalny żart? Zmyślny sabotaż? Przebiegłe zabezpieczenie przed nieautoryzowanym uruchamianiem? A może ktoś celowo zadbał o to, żeby w niepowołanych rękach system działał nie do końca poprawnie? Sensacyjne hipotezy można by mnożyć, ale powstrzymajmy póki co fantazję i spróbujmy przeanalizować fakty. Bezpośrednie sąsiedztwo kodu odpowiedzialnego za zapis wygląda następująco:

```
       BLOK:
0x136b:     RJ   r4, GENAN
0x136d:     PW   r1, [BLPASC+r5]
0x136f:     LW   r1, [BAR+r5]
0x1371:     BB   r1, 1\9
0x1373:     UJS  TPRI+3
0x1374:     RJ   r4, GENAN         (4)
0x1376:     LWT  r4, 062
0x1377:     NR   r4, r1
0x1378:     CWT  r4, 042
0x1379:     BLC  ?E
0x137a:     PW   r6, r1            (3)
0x137b:     UJS  TPRI+3
       TPRI:                       (1)
0x137c:     LW   r3, [BLPASC+r5]
0x137e:     IRB  r3, BLOK          (2)
```

Wszystko zaczyna się w procedurze obsługi wywołania systemowego *wczytaj rekord* (1), realizującego czytanie ze strumienia. Dlaczego akurat tam? Diabli wiedzą, ale póki co nie zaprzątajmy sobie tym głowy. Ważne, że następuje w niej warunkowy skok do procedury *BLOK* (2), która w przedostatniej instrukcji (3) dokonuje warunkowo felernego zapisu. Zapis ten, prócz tego, że sam w sobie jest dziwny, jest też podejrzany z punktu widzenia ścieżki wykonywania kodu:

1. Zawartość rejestru *r6*, który przechowuje zapisywaną wartość, nie jest ustawiana przez system operacyjny. Jest w nim to, co przed wywołaniem systemowym zostawił tam proces użytkownika. Według dokumentacji wywołanie to nie używa rejestru *r6* do przekazywania argumentów, więc dlaczego system operacyjny miałby z rejestru *r6* w ten sposób korzystać?
2. Zawartość rejestru *r1*, przechowującego docelowy adres zapisu, ustawiana jest w procedurze *GENAN* (4), wyglądającej ze wszech miar jak generator liczb pseudolosowych.

Błąd?

Dobrze zatem. Skoro mamy jedną linię asemblera, z którą związane są aż trzy wyjątkowo dziwne obserwacje, to załóżmy przez chwilę, że faktycznie mamy do czynienia z błędem. Spróbujmy uruchomić CROOK-a z małą poprawką: Zablokujmy podejrzany zapis przez proste zastąpienie instrukcji pod adresem *0x137a* instrukcją pustą (*NOP)*. Rezultat? W tak „naprawionym" CROOK-u zarówno kompilator, jak i inne nie działające wcześniej programy funkcjonują poprawnie.

Za każdym razem. Bez najmniejszych efektów ubocznych.

Ale zanim ktoś zdąży zakrzyknąć: *„Czyli ewidentny błąd w CROOK-u!"*, to napiszę o kolejnej obserwacji: Procedura *BLOK* występuje we *wszystkich* zachowanych wersjach jądra systemu, w postaci nie zmienionej *ani o jotę*. Wygląda na to, że nie jest to jakaś przypadkowa linia, która wkradła się z jedną, nieudaną edycją, a zamierzone, utrzymywane z wersji na wersję zachowanie systemu.

Jeśli nie błąd, to co?

Sprawa zaczyna się robić coraz ciekawsza, co przy okazji nie ułatwia utrzymywania wyobraźni w ryzach. Ach, te chwytliwe nagłówki: *„System operacyjny z Politechniki Gdańskiej odmawia wykonywania binariów z Uniwersytetu Warszawskiego!"*, *„Wielka zemsta autorów CROOK-a!"*.

Żarty na bok, rzetelna analiza na front. Skoro bezpośrednie sąsiedztwo źródła problemu nie wyjaśnia wiele, to poszukajmy krok dalej, w samej procedurze *GENAN*.

1. Czy jest ona używana gdzieś indziej? Nie. Mamy więc w CROOK-u generator liczb pseudolosowych, używany *wyłącznie* do wybrania adresu, pod który system dokonuje niszczącego zapisu.
2. Jakie inne punkty zaczepienia daje ciało procedury? Raczej mizerne:
   - mamy tam zmienną *LAST* opisaną w komentarzu jako „GENERATOR", co potwierdzałoby istnienie generatora liczb pseudolosowych,
   - są wywoływane dwie inne procedury, wyglądające na pomocnicze: *GENOB* i *GENAD*, prowadzące donikąd.

Tyle. Żadnych dalszych punktów zaczepienia. Spójrzmy więc na kod w bezpośrednim otoczeniu generatora, może się poszczęści. Dwieście linii niczym nie wyróżniającego się asemblera wyżej trafia się takie coś:

```
C1=17152.    [   A1+C1]
C2=17152.    [A1:A2+C2.]
C3=17152.    [   M +C3.]
C4=17152.    [A2:A3+C4.]
C5=17152.    [   A1+C5.]
C6=17152.    [A3:A2+C6.]
CN=17152.    [GEN(NR+CN,(A3-A2)mod077)  CR5  NR+CN  PASC ]
C7=17152.    [   S +C7.] [ /S/=/S/+A2-A1 ]
```

Osiem stałych, opatrzonych dziwnym, wyjątkowo bogatym jak na CROOK-a komentarzem. Fragment zdecydowanie odstaje od reszty, ale uwagę przykuwa coś innego: powtarzająca się liczba *17152*.

Ostatni element układanki

Zobrazowany dysk MERY-400 z Politechniki Gdańskiej to około 20MB danych. Zawartość większości plików, lub przynajmniej ich rola, jest zrozumiała. Ale niektóre z nich pozostawiają w pamięci ślad: *„w zasadzie wiadomo co to jest, ale jakoś dziwnie to wygląda”*. Takie ślady okazują się czasami być przydatne wiele miesięcy później, dopełniając brakujący fragment innej układanki. Tak też było w tym przypadku.

Liczba *17152* już gdzieś, kiedyś się pojawiła. W jakimś nie do końca zrozumiałym kontekście, przy okazji prac nad czymś zupełnie innym. Poszukajmy zatem. Przeczesanie zawartości dysku pod kątem charakterystycznej liczby odnajduje makro służące do budowania jednego z programów użytkowych. Do tego programu linkowany jest obiekt budowany assemblerem GASS z następującego źródła:

```
           .UNIT       SECRET
SYSID       =          %B                  ; NUMER ZEGARA
           .USE        CODE
$          .XVAR       1
SECRET_M   .RES        2
           .USE        DATA
           .IMP        .croota,mkmain
           .EXP        IC0,SECRET_M,main
C           =          17152
           .DATA       A1+C
A1         .DATA       A2+C,SECRET_M+C
S          .DATA       0
A2         .DATA       A3+C,A1+C
A3         .DATA       A2+C,SYSID+C,S+C

           .RES        2
IC0         STRA       $(4)
            SET    R4, #$
            CALL   R5, .croota
main        CALL   R5, mkmain
           .END        IC0
```

Zbieżność nazw symboli i stałych pomiędzy programem użytkowym a jądrem systemu, operacje arytmetyczne odpowiadające tym w komentarzach źródła systemu – to nie może być przypadek. Lokalizacja w pobliżu funkcji *main()* również nie jest przypadkowa. Do tego podejrzany generator liczb pseudolosowych i dziwnie zachowujące się programy. Fragmenty układanki zaczynają pasować do siebie idealnie. I choć dookoła dużo jeszcze pustych przestrzeni, których wypełnienie wymagać będzie sporo pracy, to można bez cienia wątpliwości stwierdzić:

W połowie lat '80, kiedy nie nazwany jeszcze nawet DRM zaczynał dopiero nieśmiało raczkować, stworzony w Instytucie Okrętowym Politechniki Gdańskiej, działający na MERZE-400 CROOK-5

dysponował *wbudowanymi w system operacyjny* mechanizmami pozwalającymi twórcom oprogramowania chronić je przed nieautoryzowanym uruchamianiem. Mechanizmami nie wspomnianymi w żadnej dokumentacji.

Pochodzenie

Jak się okazuje, historia odnalezionych zabezpieczeń sięga dalej, zarówno w czasie jak i przestrzeni, niż piąta wersja systemu operacyjnego z Politechniki Gdańskiej.

Wszystko zaczęło się w Instytucie Informatyki Uniwersytetu Warszawskiego. Pracujące tam MERY-400 działały pierwotnie pod kontrolą systemu operacyjnego SOM-3. Z czasem, kiedy rozwijane w Instytucie oprogramowanie dojrzało do stanu, w którym można je było oferować odpłatnie potencjalnym klientom, pojawił się problem: jak radzić sobie z nielegalnymi kopiami? Rozwiązaniem stała się modyfikacja SOM-3, implementująca mechanizmy uniemożliwiające poprawną pracę pirackich wersji chronionych programów.

Rosnąca popularność dojrzałego już wtedy i znacznie nowocześniejszego niż SOM-3 systemu CROOK spowodowała, że narzędzia powstałe w IIUW musiały w końcu zostać przeniesione i na tę platformę. Warunkiem koniecznym było jednak zachowanie istniejących już zabezpieczeń. I tak, po rozbudowaniu oryginalnego rozwiązania o nowe elementy oraz ustaleniu przez zespoły z UW i PG jego ostatecznego kształtu, CROOK wzbogacił się o mechanizm, którego złożoność wygląda imponująco nawet po 30 latach.

Konstrukcja systemu zabezpieczeń

Trzeba na wstępie zaznaczyć, że do realizacji celu nie zostały użyte metody kryptograficzne. Ograniczona moc obliczeniowa powodowała, że jedyną praktyczną linią obrony była wtedy niejawność użytych rozwiązań. Według dzisiejszych standardów takie *security by obscurity* nie jest uznawanym sposobem zapewniania bezpieczeństwa. Należy jednak pamiętać, że dostępne ówcześnie metody i narzędzia analizy były mniej zaawansowane, przepływ informacji utrudniony, a ilość systemów komputerowych nieporównywalnie mniejsza. To powodowało, że zastosowana protekcja przez długi czas mogła spełniać swoje zadanie.

System zabezpieczeń używany przez CROOK-5 składa się w swoim ostatecznym kształcie z następujących elementów:

1. Sprzętowy identyfikator maszyny, unikalny dla każdego systemu MERA-400 (MX-16), na którym instalowany był CROOK.
2. Zbiór zasad opisujący jak powinien zostać przygotowany program binarny mający wykorzystywać zabezpieczenia.
3. Wbudowany w system loader zabezpieczonych programów.
4. Wbudowane w system mechanizmy uniemożliwiające korzystanie z nieautoryzowanych kopii programów.
5. Narzędzie, za pomocą którego binaria autoryzowane były dla maszyny o danym identyfikatorze.

Warto w tym miejscu zwrócić uwagę na to, jaki powyższy zbiór sugeruje podział na „obszary wtajemniczenia” pomiędzy poszczególnymi uczestnikami procesu zabezpieczania programów. Było to jedną z metod realizacji wspomnianej wcześniej niejawności:

- Twórcy systemu operacyjnego nie potrzebowali identyfikatorów sprzętowych maszyn klientów, aby przygotować dla nich system. Mogły one być przechowywane w innym ośrodku i tam niezależnie zarządzane.
- Programiści również nie potrzebowali (i nie powinni byli znać) identyfikatorów maszyn klientów, do których było dystrybuowane oprogramowanie. Nie znali też algorytmu autoryzacji binariów i nie mieli środków (narzędzi) do tego celu.
- Jedynie „centrum autoryzacji” dysponowało identyfikatorami maszyn oraz narzędziem autoryzacyjnym, co pozwalało na przygotowanie kopii programu przeznaczonej dla konkretnego klienta.
- Klient końcowy nie był świadomy faktu zabezpieczenia programu (nie podawał np. żadnego klucza podczas instalacji). Nie wiedział też o istnieniu sprzętowego identyfikatora. Jeśli natomiast próbował użyć programu nie przeznaczonego dla jego maszyny, zabezpieczenie manifestowało się w sposób sugerujący raczej niesprawność systemu lub oprogramowania, niż swoje istnienie.

Identyfikator systemu

Pierwszym, kluczowym elementem mechanizmu jest identyfikator systemu. W zmodyfikowanym systemie SOM-3 był on zapisany w ciele systemu operacyjnego i różnił się między kopiami dostarczanymi użytkownikom końcowym. Pozwalał więc rozróżnić kopie SOM-3, ale nie maszyny, na których system był uruchamiany. W CROOK-5 identyfikator jest natomiast sprzętowy, a więc związany z konkretnym egzemplarzem komputera. Istnieją dwie implementacje rozwiązania i w zależności od lokalizacji identyfikator potocznie nazywany był „numerem procesora” lub „numerem zegara”. Obie wersje zrodziły się w Instytucie Okrętowym Politechniki Gdańskiej i tam też powstał prototyp pierwszej z nich. W systemie operacyjnym CROOK-5 identyfikator dostępny jest w przestrzeni jądra pod zmienną nazwaną w źródłach *PROCNU*.

Numer zegara

W nieco starszej wersji rozwiązania, identyfikator schowany jest w zegarze czasu rzeczywistego. Urządzenie to, instalowane jako kolejna jednostka w kanale znakowym, dysponuje również pamięcią PROM, na której zapisany jest bootloader pozwalający „ściągnąć” system operacyjny z dysku do pamięci komputera (poprzez mechanizm wczytywania binarnego). Tuż za znacznikiem końca kodu bootloadera zapisany jest identyfikator systemu, zwany (ze względu na swoją lokalizację) „numerem zegara”.

Po uruchomieniu systemu CROOK-5 za pomocą takiego bootloadera, sprzętowy licznik adresujący kolejne bajty w pamięci PROM zegara nie zmienia swojej wartości. Dzięki temu, wysyłając do urządzenia kolejne rozkazy czytania, można kontynuować pobieranie danych. Identyfikator systemu zapisany jest na trzech kolejnych bajtach pamięci PROM. Ich cztery najmłodsze bity zawierają kolejne (od najstarszej) cyfry identyfikatora zapisane w kodzie BCD.

Bajt 1

P	W	-	K	a_3	a_2	a_1	a_0

Bajt 2

P	W	-	K	b_3	b_2	b_1	b_0

Bajt 3

P	W	-	K	c_3	c_2	c_1	c_0

Znaczenie poszczególnych bitów to:

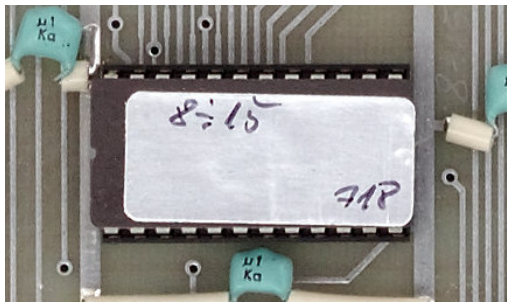
- *P* – bit parzystości
- *W* – bit ważności
- *K* – znacznik końca
- *a*, *b*, *c* – kolejne cyfry identyfikatora

Bit *K* we wszystkich bajtach musi mieć wartość *1*. Identyfikator po złożeniu ma postać:

$$PROCNU = a * 100 + b * 10 + c$$

Numer procesora

Późniejsze, ostateczne rozwiązanie używa do przechowywania identyfikatora ostatniego słowa pamięci stałej (zrealizowanej na kościach EEPROM) w pamięci MEGA Amepolu. System operacyjny wczytuje go podczas startu:



Zdj. 1: EEPROM pamięci MEGA

```
OR   r0, ?1         (2)
RW   r5, RROB+1
...
LW   r5, [-1]       (1)
...
...
LW   r4, [RROB+1]   (3)
BRC  ?1
RW   r4, PROCNU
```

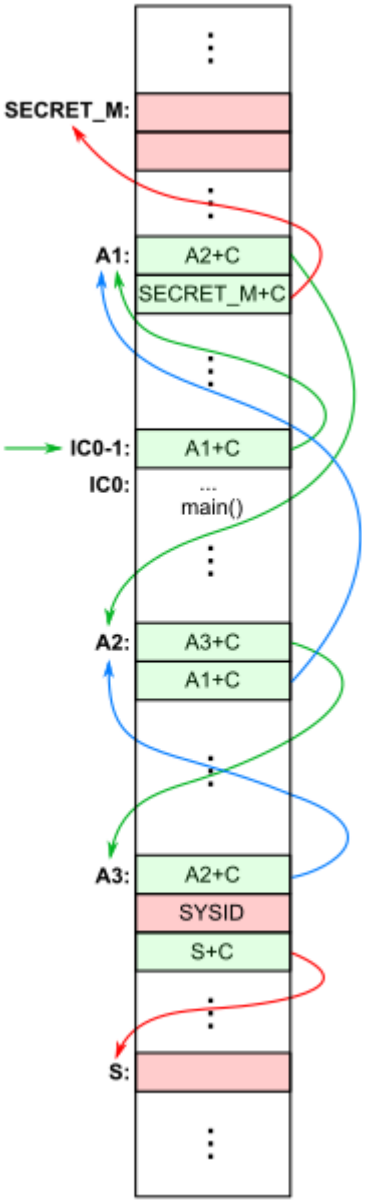
Identyfikator ładowany jest rozkazem (1) ukrytym w pętli inicjalizacyjnej pamięci. Po jej skonfigurowaniu i zasłonięciu strony pamięci stałej ustawiana jest flaga *1* w rejestrze *r0* procesora (2) i identyfikator zapamiętywany jest w obszarze roboczym. Nieco dalej w kodzie (3) wartość ta jest ostatecznie (warunkowo względem flagi *1*) przepisywana do zmiennej *PROCNU*. Takie ukrycie i rozrzucenie kodu pobierania identyfikatora jest zapewne kolejną manifestacją niejawności rozwiązania. Nie sprzyja jej natomiast inna ciekawostka - w przypadku zachowanego komputera z Politechniki Gdańskiej, identyfikator *718* zapisano również na pamięci EEPROM... pisakiem (zdjęcie 1).

W tym rozwiązaniu identyfikator był nazywany „numerem procesora”, ponieważ pamięć MEGA była elementem jednostki centralnej MJC-400, zwanej potocznie procesorem.

Zabezpieczenie programu użytkowego

Program, który ma być zabezpieczony, musi zostać wyposażony w określoną, cykliczną strukturę danych (rysunek 2). Ma ona następujące cechy:

- pozwala systemowi operacyjnemu łatwo i jednoznacznie stwierdzić, czy załadowany program jest zabezpieczony,
- pozwala systemowi zweryfikować, czy kopia programu jest autoryzowana dla danej maszyny,
- pozwala programowi użytkowemu sprawdzić, czy został legalnie uruchomiony,
- jest trudna do zauważenia i rozpoznania przez intruza analizującego kod programu.



Rys. 2: Struktura danych identyfikująca zabezpieczony program

W strukturze występują następujące elementy:

- *A1*, *A2*, *A3* – adresy w programie, pod którymi umieszczone są kolejne fragmenty struktury.
- *C* – arbitralna stała o wartości *17152*, znana zarówno systemowi operacyjnemu jak i twórcom zabezpieczanych programów, używana jako przesunięcie dla adresów. Źródła wskazują, że mechanizm był gotowy do użycia różnych stałych dla przesunięć różnych elementów struktury.
- *SECRET_M* – adres zmiennej zapisywanej przez system operacyjny, a dostępnej programowi użytkowemu, dzięki której będzie on mógł w trakcie pracy stwierdzić, czy jest uruchamiany legalnie (*=0*), czy nielegalnie (*!=0*).
- *S* – adres pola używanego przez system operacyjny podczas ładowania programu.
- *SYSID* – zmodyfikowany identyfikator systemu (sposób jego modyfikacji zostanie opisany w dalszej części).
- *IC0* – punkt wejścia do programu; adres, na który system operacyjny ustawia licznik rozkazów przed przełączeniem kontekstu na proces użytkownika.
- *IC0-1* – miejsce, które było ustalonym punktem początkowym struktury, znajdującym się zawsze bezpośrednio przed adresem startu programu.

Strzałkami zielonymi pokazane są na rysunku połączenia pozwalające skompletować całą strukturę. Znając początek struktury oraz wartość stałej *C*, można poruszać się w strukturze odczytując kolejne jej elementy. Strzałkami niebieskimi oznaczone są cykle służące do weryfikacji jej poprawności. Strzałki czerwone wskazują lokalizacje dodatkowych danych używanych podczas uruchamiania zabezpieczonego programu.

Przygotowanie struktury po stronie programisty polega na:

- ustaleniu w programie adresów *A1*, *A2*, *A3*, *SECRET_M* i *S*,
- zarezerwowaniu miejsca na poszczególne fragmenty struktury,
- wypełnieniu pól łączących fragmenty struktury (oznaczonych na rysunku 2 zielonym tłem).

Pola oznaczone kolorem czerwonym nie są przez programistę wypełniane, w szczególności nie jest wypełniane pole *SYSID*.

Lokalizacja adresów jest w zasadzie dowolna, jednak z założenia o niejawności powinny znajdować się one w odległych miejscach programu, tak, aby możliwie dobrze ukryć przed wzrokiem intruza dane pod nimi zapisane. Jedyną wymuszoną lokalizacją jest adres słowa zawierającego *A1+C*.

Struktura może być oczywiście implementowana w dowolnym języku programowania. Poniżej przykład użycia jej w programie przeznaczonym dla assemblera ASSM. Poszczególne fragmenty znajdują się w różnych miejscach pliku ze źródłem:

```
C1=17152.
C2=C1.C3=C1.C4=C1.C5=C1.C6=C1.C7=C1.
...
A1+C1.
START:LW,X(DESNAS)
...
A1:A2+C2.MM+C3.
...
A2:A3+C4.A1+C5.
...
A3:A2+C6.0.XX+C7.
...
MM:0.0.XX:0.
```

Istniał również uniwersalny moduł GASS-a dla programów łączonych konsolidatorem LINK, który pozwalał na łatwe dodanie zabezpieczenia do istniejącego programu źródłowego. Poniżej jego fragment zawierający opis struktury:

```
SECRET_M  .RES         2
...
C          =           17152
          .DATA        A1+C
A1        .DATA        A2+C, SECRET_M+C
S         .DATA        0
A2        .DATA        A3+C, A1+C
A3        .DATA        A2+C, SYSID+C, S+C
```

Przykładowa implementacja w języku C, pokazana poniżej, jest chyba najbardziej zawiła. Poszczególne definicje zostały nawet rozmyślnie rozrzucone po różnych plikach źródłowych:

```
#define _HC1 17152
int *_HA11[2] = {&_HA22[_HC1], &_HMM[_HC1]};
int *_HA22[2] = {&_HA33[_HC1], &_HA11[_HC1]};
int *_HA33[4] = {&_HA22[_HC1], 2348, &_HA33[3+_HC1], 2349};
int _HMM[2] = {2348, 2348};
static int *_HA00[2] = {&_HA11[_HC1], &_HA11[_HC1]};
```

Ładowanie zabezpieczonego programu

Serce mechanizmu zabezpieczeń bije w systemie operacyjnym. Jego działanie rozpoczyna się tuż po załadowaniu programu z dysku do pamięci, jeszcze przed jego uruchomieniem. Do obsługi zabezpieczonych programów CROOK-5 używa dwóch informacji zapisanych w strukturze każdego procesu:

- Bit 9 pola *BAR* mówi o tym, czy kopia programu jest nieautoryzowana. Bit jest zapalany dla kopii nielegalnych, gaszony dla programów niezabezpieczonych oraz kopii legalnych. Przed weryfikacją programu bit jest domyślnie gaszony.
- Pole *BLPASC* mówi o tym, czy podczas wykonywania programu należy podjąć kroki przeszkadzające użytkownikowi w korzystaniu z kopii nielegalnej. Przed weryfikacją programu w polu wpisywana jest wartość *-1*, która oznacza zaniechanie podejmowania takich czynności.

Każdy ładowany program traktowany jest jako potencjalnie zabezpieczony i jest dla niego wykonywana procedura weryfikacji struktury opisanej w poprzednim rozdziale. Poszczególne jej kroki to:

- załaduj zawartość *IC0-1* (=*A1*+*C*) i oblicz *A1*,
- załaduj zawartość *A1* (=*A2*+*C*) i oblicz *A2*,
- załaduj zawartość *A2* (=*A3*+*C*) i oblicz *A3*,
- załaduj zawartość *A2+1* (=*A1*+*C*) i sprawdź poprawność *A1*,
- załaduj zawartość *A3* (=*A2*+*C*) i sprawdź poprawność *A2*,
- załaduj zawartość *A3+1* (=*SYSID*) i wartość zapisz tymczasowo w rejestrze,
- załaduj zawartość *A3+2* (=*S*+*C*), oblicz *S*, załaduj wartość znajdującą się pod tym adresem, dodaj do niej różnicę *A2-A1* i zapisz ponownie pod *S*.

Dla programów nie posiadających zabezpieczenia, niemal na pewno przynajmniej jeden z kroków weryfikacyjnych się nie powiedzie. W takim wypadku program traktowany jest jako legalnie pozyskany i na takich zasadach uruchamiany. Jeśli natomiast program zostanie rozpoznany jako zawierający zabezpieczenie, następuje jego weryfikacja.

Rozpoczyna się ona od przygotowania dwóch wartości. Pierwsza z nich, nazwijmy ją *X*, wywodzona jest z numeru procesora zapisanego w systemie operacyjnym:

$$X = (PROCNU + C) \ll 1$$

Przesunięcie o jeden bit w lewo potrzebne jest do poprawnej pracy algorytmu opisanego dalej, ma jednak skutek uboczny: ogranicza efektywną przestrzeń identyfikatorów systemu do 32768 wartości.

Druga wartość, *Y*, wywodzona jest z różnicy adresów w strukturze zabezpieczającej program:

$$Y = ((A3 - A2) \bmod 63) + 56$$

Zauważalnym zamierzeniem autorów było uzyskać liczbę, która zależy od umiejscowienia pozycji *A2* i *A3* względem siebie (będzie się więc różnić się między programami), ale jest stosunkowo mała

(w praktyce od 56 do 119).

Następnie na przygotowanym argumencie *X* wywoływana jest *Y* razy funkcja mieszająca *GEN()*, na którą składają się następujące kroki:

- policz różnicę symetryczną 1 i 3 bitu argumentu (przy numeracji bitów 15-0),
- przesuń argument o 1 w prawo,
- wpisz bit z pierwszego kroku na najstarszą pozycję argumentu.

Wynikiem operacji jest wielokrotnie wymieszany, powiększony uprzednio o stałą *C* numer procesora.

Wartość ta odejmowana jest w kolejnym kroku od wartości *SYSID* pobranej uprzednio z ciała uruchamianego programu. Jeśli różnica jest niezerowa, oznacza to, że numer procesora maszyny jest niezgodny z tym, dla którego przygotowana została kopia programu i zostaje ona uznana za nielegalną. To pokazuje też, że *SYSID* zapisane w zabezpieczonym programie nie jest wprost numerem procesora, a numerem poddanym takiej samej operacji mieszania, jaką wykonuje CROOK-5. Jest to oczywiście kolejnym przejawem założeń o niejawności. Wniosek ten stał się też śladem prowadzącym do konieczności autoryzacji programów opisanej w dalszym rozdziale.

W przypadku stwierdzenia nielegalności kopii programu system wykonuje następujące czynności:

- bit 9 w polu *BAR* procesu jest zapalany,
- w polu *BLPASC* zapisywana jest losowa wartość różna od *-1*.

Niezależnie od wyniku weryfikacji w polu *SECRET_M* programu zapisywana jest arytmetyczna negacja obliczonej różnicy identyfikatorów. Oznacza to, że program użytkowy może sprawdzić, czy został uruchomiony na właściwej dla niego maszynie, poprzez porównanie *SECRET_M* z zerem, a w przypadku niezgodności podjąć dodatkowe akcje. Przykładem jest edytor tekstów EXM, którego nielegalne kopie odmawiają zapisywania zbiorów.

Ograniczenie działania nielegalnej kopii programu

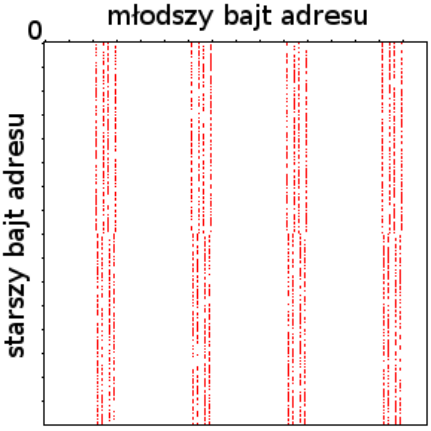
Oprócz akcji, które w przypadku stwierdzenia nielegalnego uruchomienia program użytkowy mógł wykonać we własnym zakresie, CROOK-5 sam podejmuje czynności przeciwko nieautoryzowanym kopiom. Mechanizm ten uruchamiany jest zawsze podczas wołania ekstrakodów *INPR* i *PINP*. Realizują one funkcje wywołania systemowego czytania ze strumienia, a więc będą w programie niemal na pewno używane. Działanie mechanizmu jest następujące:

1. Jeśli pole *BLPASC* w strukturze procesu ma wartość *-1*, kontynuuj standardową obsługę ekstrakodu. W przeciwnym wypadku kontynuuj działanie zabezpieczenia.
2. Pobierz wartość z generatora liczb pseudolosowych (różną od *-1*) i zapisz ją pod *BLPASC*.
3. Jeśli bit 9 w polu *BAR* struktury procesu jest zgaszony, kontynuuj standardową obsługę ekstrakodu. W przeciwnym razie kontynuuj działanie zabezpieczenia.
4. Pobierz liczbę *P* z generatora liczb pseudolosowych
5. Jeśli spełniony jest warunek $(P \wedge 110010) = 100010$ to potraktuj *P* jako adres w przestrzeni procesu wołającego ekstrakod i zapisz pod pod nim zawartość rejestru *r6*.

Ostatni warunek mówi, że:

- Na każdą stronę zaalokowaną w segmencie pamięci przeznaczonej dla procesu przypadało 16 komórek pamięci, których zawartość mogła być zniszczona. A to znów oznacza, że niezależnie od rozmiaru programu mógł on zostać uszkodzony.
- Prawdopodobieństwo, że wywołanie czytania ze strumienia zniszczy komórkę pamięci w przestrzeni procesu wynosiło $\frac{\textit{ilość stron pamięci procesu}}{4096}$. Czyli programy większe były bardziej narażone na skutki działania zabezpieczenia.

Rysunek 3 pokazuje w jakie obszary w przestrzeni procesu system operacyjny trafia losowymi adresami. Generator liczb pseudolosowych czerpał entropię z informacji o tym, w jakim momencie okresu pracy planisty systemu operacyjnego nastąpiło wywołanie systemowe czytania ze strumienia. Ocena jakości generatora nie ma sensu w warunkach emulacji, ponieważ parametry czasowe pracy takiego systemu odbiegają znacznie od prawdziwej maszyny.



Rys. 3: Obszary pamięci procesu atakowane przez zabezpieczenie w systemie CROOK-5

Podsumowując: Nielegalne kopie programów uruchomione w systemie CROOK-5 są w pamięci powoli, sukcesywnie niszczone. Rezultaty takiego niszczenia są na tyle przypadkowe i różnorodne, że nieświadomy zabezpieczeń użytkownik będzie najprawdopodobniej podejrzewał niesprawność systemu komputerowego lub błąd w programie. Ale na podejrzeniach jego możliwości się kończą, bo do autorów „wadliwego” programu raczej się o pomoc nie zgłosi. Kopia programu na dysku pozostaje oczywiście nienaruszona.

Autoryzacja programów

Umieszczenie stosownej struktury danych w programie jest tylko przygotowaniem terenu. Obserwacja zachowań systemu operacyjnego pokazuje, że pole *SYSID* musi jeszcze zostać w specyficzny sposób wypełnione. Służy do tego dodatkowe narzędzie: UZDAT.

Wywołanie go bez parametrów pozwala sprawdzić, jaki jest numer procesora maszyny:

```
`uzdat
NUMER PROCESORA : 718
```

Zachowanie to jest o tyle ciekawe, że identyfikator systemu nie jest informacją dostępną dla programu użytkowego wprost. Narzędzie używa sztuczki korzystającej z faktu, że system operacyjny w trakcie ładowania zabezpieczonego programu zapisuje pod *SECRET_M* różnicę między numerem systemu zapisanym w programie, a znanym sobie, uprzednio wymieszanym numerem procesora. UZDAT jest programem z zabezpieczeniem – zawiera strukturę danych go identyfikującą, ale w polu *SYSID* wpisane ma *0*. *SECRET_M* jest więc po uruchomieniu programu ustawione przez system operacyjny na *SYSID* wymieszane funkcją *GEN()*. Ponieważ funkcja ta jest odwracalna i programowi znane są wartości *C*, *A3* i *A2*, UZDAT może z *SYSID* obliczyć faktyczny numer procesora.

Pozostaje jeszcze pytanie dlaczego CROOK-5 nie niszczy UZDAT, skoro ten ma wpisany niepoprawny, zerowy identyfikator systemu? Autor narzędzia, znając oczywiście zachowanie CROOK-a, zabezpieczył się przed jego niszczycielskimi zapędami używając do czytania danych wyłącznie ekstrakodów transmisji blokowych, które nie wyzwalają procedury nadpisującej zawartość pamięci procesu.

Podobne użycie odwrotności funkcji *GEN()* ma miejsce w kolejnym zastosowaniu narzędzia. Podając jako parametr nazwę zbioru z zabezpieczonym programem można sprawdzić, dla jakiej maszyny został on autoryzowany:

```
`uzdat cc00
NUMER ZE ZBIORU : 718
```

Jeśli program nie jest zabezpieczony, narzędzie o tym poinformuje:

```
`uzdat stat
*PROGRAM NIEPRZYGOTOWANY
```

Wreszcie, odpowiednio przygotowany program można „uzdatnić” do uruchomienia na maszynie o numerze procesora podanym jako drugi argument:

```
`uzdat cc00 1234
PRAWIDLOWO
`uzdat cc00
NUMER ZE ZBIORU : 1234
```

Wektory ataku

Skoro znane już są wszystkie szczegóły dotyczące mechanizmu zabezpieczeń używanego przez system CROOK-5, można pokusić się o podsumowanie możliwych wektorów ataku (z uwzględnieniem środków dostępnych w latach '80 użytkownikowi MERY-400).

Identyfikator systemu

Atak na identyfikator systemu wymaga najmniej zaangażowania intelektualnego, ale jest jednocześnie najmniej elastyczny. Zakładając, że komuś udałoby się uzyskać wiedzę o istnieniu i lokalizacji numeru procesora lub numeru zegara, mógł on przygotować „klon” innego komputera i z powodzeniem uruchamiać na nim oprogramowanie zakupione dla klonowanej maszyny. Choć mało elastyczne, rozwiązanie to nie jest pozbawione praktycznego zastosowania - dwóch lub więcej użytkowników systemu mogło „umówić się” i dzielić kosztami oprogramowania. Atak ten ma jedną silną zaletę: nie wymaga absolutnie żadnej wiedzy na temat tego, jak mechanizm zabezpieczeń naprawdę działa.

W praktyce atak wymaga podjęcia różnych kroków dla różnych lokalizacji identyfikatora. W przypadku numeru procesora zapisanego w pamięci MEGA należy ponownie zaprogramować kości pamięci EEPROM, co jest procedurą stosunkowo prostą w realizacji. W przypadku zegara czasu rzeczywistego operacja jest o tyle utrudniona, że kości PROM są jednokrotnie programowalne. Trzeba więc zakupić nowe kości pamięci, a do ich zaprogramowania użyć narzędzia potencjalnie trudniej dostępnego niż tego dla pamięci EEPROM.

System operacyjny

Ataki na system operacyjny wymagają największej wiedzy, są najtrudniejsze w realizacji, ale dają najlepsze rezultaty. Raz złamany CROOK-5 zawsze już uruchomi dowolny program, przeznaczony dla dowolnego innego komputera. Można rozpatrzyć dwa różne podejścia. W obu przypadkach atakujący musi przede wszystkim zdeasemblować i przeanalizować jądro systemu. O ile deasemblacja nie stanowi problemu, a analiza kodu, choć żmudna, ma ten sam poziom skomplikowania co i dziś, o tyle śledzenie działania systemu wymaga podejścia do niego z zewnątrz. A jedyną w praktyce dostępną metodą, która pozwalała na to w latach '80, było użycie pulpitu technicznego MERY-400. To czyni z zadania iście syzyfową pracę.

Dodatkowo, po odpowiednim „poprawieniu" systemu operacyjnego czeka na intruza jeszcze jedna niespodzianka. CROOK-5, zarówno podczas startu, jak i cyklicznie, w trakcie działania, sprawdza swoją sumę kontrolną. A z niewłaściwą sumą kontrolną odmówi uruchomienia się. Ostatnim krokiem musi więc być odnalezienie gdzie i jak liczona jest wspomniana suma i albo jej poprawienie, albo zablokowanie również tego mechanizmu.

Atak można przypuścić z dwóch stron:

1. **Kod ładujący program.** W procedurze ładowania programu użytkowego można zapewnić, aby niezależnie od wyniku testu poprawności numeru procesora, do pola *BLPASC* procesu zapisywana była zawsze wartość *-1*, a bit 9 pola *BAR* zawsze pozostawał zgaszony. Dodatkowo trzeba też oszukać sam program, przekonując go, że został uruchomiony na właściwej maszynie. Wymaga to wpisania *0* do pola *SECRET_M* w jego przestrzeni adresowej.
2. **Kod niszczący proces.** W procedurze obsługi wywołania systemowego czytania ze strumienia można zablokować zupełnie podejmowanie akcji dla nielegalnie uruchomionych kopii programów. Do zupełnego wyłączenia zabezpieczeń trzeba by jednak jeszcze, podobnie jak poprzednio, zapisać wartość *0* do *SECRET_M*.

Program użytkowy

Atak na program użytkowy jest o tyle prostszy od ataku na jądro CROOK-a, że śledzenie wykonywania procesu w systemie można przeprowadzić za pomocą debuggera DEBU dostarczanego wraz z CROOK-iem, co jest znacznie efektywniejsze, niż używanie pulpitu technicznego. Problem polega na tym, że bez żadnej podpowiedzi trudno jest odnaleźć sprytnie schowaną między linijkami programu strukturę danych, która w najlepszym wypadku używana jest jednokrotnie, a i to tylko w jednym jej punkcie.

Ostatecznym rezultatem ataku jest natomiast złamany program, który nie tylko jest już na zawsze gotowy do użycia na danej maszynie, ale zadziała również na innych komputerach. Po niewątpliwie czasochłonnym rozgryzaniu zabezpieczeń jednego programu użytkowego dostaje się też przepis, który bez wysiłku pozwala odblokowywać kolejne programy. Wystarczy bowiem tylko:

- nadpisać słowo *IC0-1* dowolną, różną od znajdującej się tam, wartością,
- zapisać w *SECRET_M* wartość *0*.

Narzędzie autoryzacyjne

A jeśli już rozpatrywać wszystkie wektory ataków, to absolutnie najskuteczniejszą i wymagającą najmniej pracy umysłowej metodą jest zdobycie kopii narzędzia UZDAT. Inżynieria społeczna w latach '80 była z pewnością nie mniej skuteczna niż dziś.

Jakub Filipowicz, http://mera400.pl

Wrocław, marzec 2016