Podatności aplikacji desktopowych

Przegląd podatności oraz stosowanych zabezpieczeń w aplikacjach desktopowych, zarówno klienckich jak i serwerowych, działających jako typowe pliki wykonywalne.

Autor: Grzegorz Niemirowski, www.grzegorz.net

Spis treści

Wstęp

Złożoność współczesnego oprogramowania oraz mała świadomość zagadnień bezpieczeństwa wśród programistów, przyczyniają się do istnienia wielu dziur w bezpieczeństwie aplikacji oraz ich różnorodności. W tym opracowaniu skupiono się na najczęściej występujących podatnościach a także stosowanych sposobach mających ograniczyć ich występowanie lub możliwość wykorzystania.

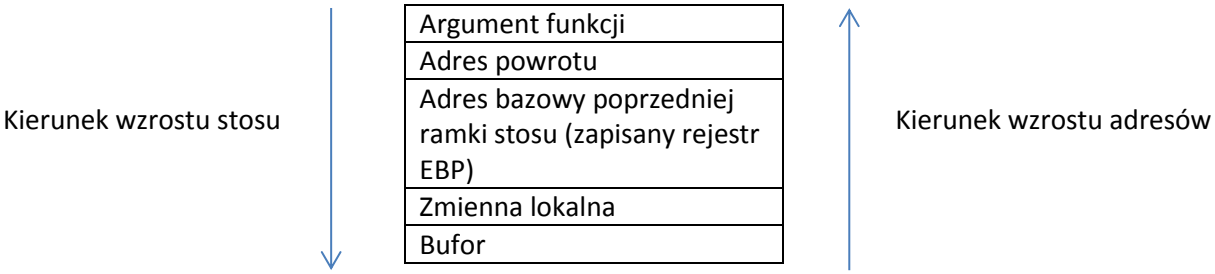
Ataki na aplikacje opierają się głównie na wykorzystaniu sytuacji nietypowych, nieprzewidzianych przez programistę. Może to być dostarczenie zbyt dużej ilości danych, dostarczenie danych nieprawidłowych czy też wykorzystanie specyfiki zachowania systemu operacyjnego. Z tego względu przeprowadzenie ataku często nie jest trywialne, nie tylko trzeba znaleźć błąd w programie ale także opracować strategię wykorzystania go do zrealizowania swojego celu, np. wykonania swojego kodu albo doprowadzenia do niedostępności określonej usługi (atak DoS). Jednocześnie niełatwym zadaniem jest pisanie bezpiecznego kodu, który zapewniłby stuprocentową odporność aplikacji na ataki. Ponadto sytuacja zmienia się w czasie: znajdowane są wektory ataku, pojawiają się zabezpieczenia, zmieniają się używane techniki tworzenia aplikacji a także narzędzia wykorzystywane przez atakujących. Z drugiej strony użytkownicy często zaniedbują łatanie podatnych aplikacji a programiści powielają stare błędy. Sytuacja jest więc złożona i na powodzenie ataku składa się wiele czynników. Dlatego w praktyce konkretny scenariusz ataku może się powieść tylko na dosyć ograniczonej ilości systemów, posiadających określoną, podatną wersję aplikacji, nieużywających pewnych zabezpieczeń i skonfigurowanych w odpowiedni sposób. W momencie udostępnienia poprawki przed producenta dziurawego programu liczba podatnych komputerów zaczyna się jeszcze bardziej zawężać.

Artykuł ten nie opisuje szczegółowo żadnego schematu włamania od początku do końca. Daje jednak ogólną orientację w kwestii bezpieczeństwa aplikacji i pozwala na rozpoczęcie samodzielnego poszukiwania podatności oraz metod ich wykorzystania. Przedstawia najpopularniejsze metody ataku i zabezpieczenia, zawiera także praktyczne wskazówki przydatne przy pisaniu exploitów.

Przepełnienie bufora na stosie

Przepełnienie bufora znajdującego się na stosie jest jedną z najczęściej stosowanych technik pozwalających na wykonanie przez aplikację kodu dostarczonego przez atakującego. Stos jest obszarem pamięci, na którym przechowywane są argumenty wywołania funkcji, adres powrotu, zachowany adres ramki stosu funkcji wywołującej a także lokalne zmienne automatyczne funkcji. Generalnie stos działa jak kolejka LIFO. Procesory x86 zapewniają obsługę stosu poprzez instrukcje PUSH (do umieszczenia wartości na stosie) oraz POP (do zdjęcia wartości ze stosu). Położenie szczytu stosu pamiętane jest w rejestrze ESP (Stack Pointer). Ważną rolę pełni też rejestr EBP (Base Pointer), który przechowuje adres bazowy ramki stosu danej funkcji. Przy wywołaniu funkcji, w jej prologu, wartość EBP funkcji wywołującej odkładana jest na stosie a do rejestru EBP wpisywana jest wartość z rejestru ESP. W ten sposób następuje przesunięcie ramki stosu. Rejestr EBP wykorzystywany jest m.in. przy dostępie do zmiennych lokalnych znajdujących się na stosie. Aplikacja odwołuje się do nich liniowo, przez offset względem adresu zapisanego w EBP, bez uciekania się do typowych dla stosu instrukcji POP i PUSH.

Poniżej przedstawiono typowy wygląd szczytu stosu po wywołaniu funkcji. Widoczny jest argument odłożony przez funkcję wywołującą oraz adres powrotu odłożony na stosie przez procesor podczas wykonywania skoku. Następnie na stosie znajduje się rejestr EBP zapisany przez wywoływaną funkcję, aby mógł zostać przywrócony przy jej zakończeniu. Następnie na stosie umieszczane są zmienne automatyczne (lokalne) danej funkcji, w tym np. zmienna tablicowa pełniąca rolę bufora.



Na czym polega atak z wykorzystaniem przepełnienia bufora? Wykorzystywany jest tu fakt, że bufor znajdujący się na stosie, sąsiaduje z innymi zmiennymi oraz zapisanym adresem powrotu z funkcji. Jeśli aplikacja nie kontroluje ilości danych zapisywanych w buforze, możliwe jest nadpisanie danych z nim sąsiadujących. Czasami przydatne może być nadpisanie zmiennych w celu zmiany zachowania aplikacji. Znacznie częściej jednak celem jest nadpisanie adresu powrotu do funkcji wywołującej. Powrót z funkcji odbywa się poprzez wywołanie instrukcji RET. Instrukcja ta zdejmuje ze stosu zapisany adres powrotu i skacze do niego wpisując go do rejestru EIP (Instruction Pointer). Tak więc przepełniając bufor i nadpisując adres powrotu można spowodować, że procesor skoczy w zupełnie inne miejsce. Sama kontrola EIP jeszcze jednak niewiele daje, chodzi bowiem o to, żeby móc wykonywać dowolne instrukcje. Skoro jednak atakujący kontroluje zawartość bufora, może umieścić w nim kod wykonywalny a następnie umieścić w EIP adres jego początku. W ten sposób atakujący może zmusić aplikację do wykonania dowolnego kodu. Często kod ten ma zapewnić dostęp do powłoki zdalnego systemu, dlatego też określany jest nazwą shellcode.

Warto zauważyć, że przepełnienie bufora powoduje nadpisanie adresu powrotu ponieważ na architekturze x86 stos rośnie w dół, tzn. kolejne odkładane na nim elementu znajdują się w pamięci pod coraz niższymi adresami. Elementy bufora adresowane są jednak standardowo. W ten sposób przepełniając bufor i nadpisując wyższe adresy, nadpisuje się elementy stosu znajdujące się w nim niżej, czyli odłożone najwcześniej, a więc właśnie takie jak zapamiętana wartość rejestru EBP, adres powrotu czy argumenty wywołania funkcji. Tutaj można sobie zadać pytanie, czy na architekturach, gdzie stos rośnie w górę (lub też w inny sposób) występują podatności związane z przepełnieniem bufora znajdującego się na stosie. Okazuje się, że tak, podatności te są prawie identyczne i kierunek rośnięcia stosu nie ma tu większego znaczenia.

Uwagi praktyczne

Ogólnie atak może się wydawać prosty: trzeba przesłać do aplikacji odpowiednio dużą paczkę danych, zawierającą shellcode oraz jego adres tak, aby adres ten znalazł się w EIP. W praktyce nie jest to jednak takie łatwe, trzeba wziąć pod uwagę kilka czynników:

- Po pierwsze trzeba najpierw oczywiście znaleźć podatność. Można w tym celu analizować kod źródłowy aplikacji, jeśli jest dostępny, lub też próbować dostarczyć aplikacji odpowiednio

spreparowane dane i obserwować jej zachowanie. Dane dostarczane są generalnie na dwa sposoby: przez sieć lub za pomocą pliku, zależnie czy jest to aplikacja sieciowa (np. serwer HTTP) czy też program pracujący z dokumentami (np. czytnik PDF). Bardzo przydatny jest tutaj fuzzing, który pozwala na automatyczne generalnie zestawów danych, wprowadzanie ich do aplikacji oraz monitorowanie jej zachowania. Oprócz stwierdzenia występowania błędu przepełnienia bufora trzeba też sprawdzić, czy bufor ten znajduje się na stosie czy na stercie. Jeśli na stercie, wówczas trzeba użyć innych technik w celu wykonania kodu.

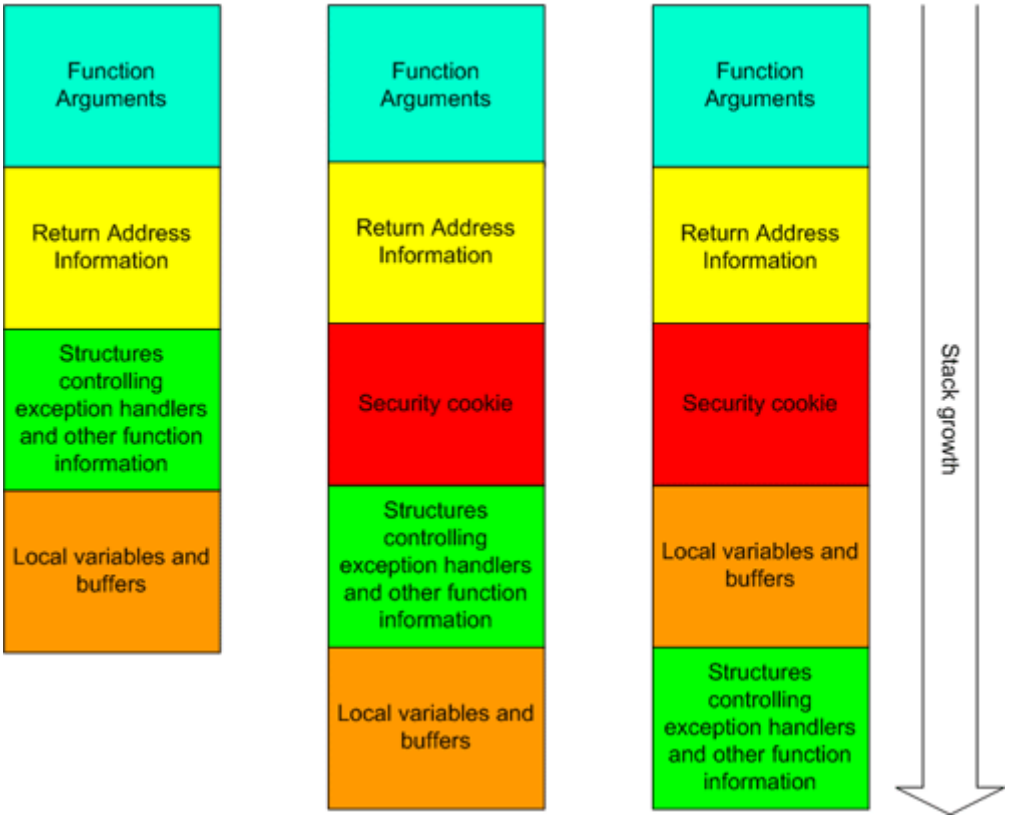
- Dane, czy to przesyłane przez sieć, czy to zawarte w pliku, mają zwykle jakąś strukturę i są parsowane. Dlatego zazwyczaj nie można oczekiwać, że znajdą się w danym buforze w całości albo w niezmodyfikowanej postaci. Należy tutaj zwrócić uwagę m.in. na bajty o wartości 0, oznaczające koniec łańcucha znaków, przejścia do nowej linii (CR, LF) oraz inne białe znaki. Zależy to od danej aplikacji i formatu danych. W związku z tym, że niektóre bajty mogą zostać odfiltrowane, trzeba zadbać aby nie było ich ani w shellcodzie ani w dostarczanym adresie, który ma znaleźć się w EIP. W pierwszym przypadku stosowane jest kodowanie shellcode'u tak, aby znalazły się w nim tylko bajty z określonego zakresu. W przypadku adresu należy także zadbać aby nie znalazły się w nim niewłaściwe bajty.
- Szczególnym przypadkiem są aplikacje, które przyjmują dane ASCII, ale wewnętrznie używają Unicode. Powoduje to duże utrudnienie dla atakującego, gdyż tylko częściowo może kontrolować zawartość bufora. Nie można bowiem utworzyć dowolnego ciągu znaków Unicode dysponując tylko znakami ASCII. Ogranicza to drastycznie zakres adresów, które można użyć w shellcode. W przypadku ataków na SEH (opisanych w dalszej części) dodatkowo ogranicza liczbę dostępnych opcode'ów skoku. Może to nawet wymusić rezygnację z instrukcji skoku, ale wtedy zawartość rekordu EXCEPTION_REGISTRATION musi nie tylko uwzględniać wymagania typowe dla ataków na SEH, uwzględniać konwersję z ASCII do Unicode, ale też być sekwencją opcode'ów, które nie należą do instrukcji, które mogłyby zaburzyć działanie exploitu, np. wykonując skok.
- Przy pisaniu exploitów praktycznie nie podaje się w dostarczanych do aplikacji danych bezpośredniego adresu bufora. Adres ten może się bowiem zmieniać np. w zależności od wersji systemu operacyjnego. Ponadto adres ten zawiera zero, co powoduje trudności opisane w punkcie poprzednim. W związku z tym do EIP wpisuje się adres miejsca w kodzie aplikacji, które zawiera kod pozwalający skoczyć do shellcode'u. W tym celu wykorzystuje się instrukcje takie jak: JMP, RET, PUSH i POP, za pomocą których pobiera się adres shellcode'u z rejestru lub ze stosu i wykonuje skok. W momencie, w którym można nadpisać adres powrotu, należy sprawdzić za pomocą debuggera, jakie wartości znajdują się w rejestrach procesora oraz na stosie, w szczególności, czy nie ma tam adresu shellcode'u. Np. jeśli adres shellcode'u znajduje się w rejestrze EAX, można wykorzystać instrukcję JMP EAX. Z kolei jeśli np. adres ten znajduje się na stosie, na miejscu drugim od góry, można wykorzystać instrukcję POP aby zdjąć najwyższy element a następnie instrukcję RET, aby skoczyć do adresu znajdującego się teraz na szczycie stosu. Trzeba się tutaj wykazać pewną dozą pomysłowości. Należy jeszcze wyjaśnić skąd wziąć te instrukcje. Otóż wystarczy przejrzeć załadowane przez aplikację moduły pod kątem potrzebnej sekwencji instrukcji, a raczej ich OPcode'ów. Trzeba więc tak naprawdę znaleźć tylko odpowiednią sekwencję bajtów, może ona należeć nawet do innych instrukcji. Ważne, żeby była w wykonywalnym obszarze pamięci i pod przewidywalnym adresem. W tym celu wykorzystuje się często systemowe biblioteki DLL.

Stack cookies

Podstawy mechanizmu stack cookies

Stack cookies to zabezpieczenie mające utrudnić napisanie exploitu bazującego na przepełnieniu bufora na stosie. Jest funkcją kompilatora, w Visual Studio wprowadzoną po raz pierwszy w wersji 2002, a od wersji 2003 jest włączona domyślnie. Jej użycie kontroluje przełącznik /GS. Ochrona przed wykorzystaniem podatności na przepełnienie bufora polega na tym, że przy uruchamianiu funkcji, na stosie, pomiędzy adresem powrotu a buforem (dokładniej między buforem a zapisaną wartością rejestru EBP) umieszczane jest tzw. stack cookie. Jest to wartość powstająca przez sXORowanie globalnej wartości __security_cookie z wartością rejestru ESP. __security_cookie to zmienna, która otrzymuje losową wartość podczas inicjalizacji modułu. Przed wyjściem z funkcji sprawdzane jest, czy stack cookie nie zostało nadpisane. Ze względu bowiem na jego położenie, atakujący nie może nadpisać na stosie adresu powrotu nie nadpisując stack cookie. Wartość stack cookie jest weryfikowana przez wywołanie funkcji __security_check_cookie(), która XORuje wartość ze stosu z wartością rejestru ESP i porównuje wynik z globalnym stack cookie. W przypadku niezgodności generowany jest wyjątek i następuje zakończenie procesu z kodem błędu STATUS_STACK_BUFFER_OVERRUN (0xc0000409).

Obsługa stack cookies nie jest wkompilowywana w każdą funkcję. Kompilator za pomocą heurystyki decyduje, czy dodać sprawdzanie czy nie. Weryfikacja stack cookie w każdej funkcji mogłaby mocno obniżyć szybkość działania aplikacji. Ponieważ język C nie ma dedykowanego typu dla łańcuchów tekstowych, przyjmowane jest, że chronione są funkcje w których występują tablice, które zajmują co najmniej 5 bajtów. Mogą to być tablice, których elementy są jedno- (ASCII) lub dwubajtowe (Unicode). Nie będą więc np. chronione funkcje, w których są tablice typu int[]. Stack cookies są także wyłączone dla funkcji, które nie są optymalizowane, mają zmienną listę argumentów, posiadają słowo kluczowe naked (wyłączające generowanie prologu i epilogu) lub też posiadają na początku wstawki asemblerowe. Chronione są za to wszystkie bufory alokowane za pomocą _alloca.



W związku z tym w kolejnych wersjach Visual Studio pojawiły się ulepszenia i optymalizacje tego procesu. W Visual Studio 2005 SP1 dodano obsługę dyrektywy strict_gs_check, która pozwalała dodawać weryfikację stack cookies bardziej agresywnie i objąć ochroną więcej funkcji. Z kolei w Visual Studio 2010 do funkcji objętych ochroną za pomocą stack cookies dodano te, które zawierają struktury. Nie dotyczy to jednak struktur, które mają pola o typie wskaźnikowym. Poszerzono też ochronę tablic, chronione są wszystkie, które nie są typu wskaźnikowego. Nadal jednak tablica musi mieć co najmniej dwa elementy. Chronione są też struktury zawierające tablice kwalifikujące się do ochrony. Jednocześnie jednak wprowadzono optymalizacje, które wyłączają z ochrony niektóre przypadki, np. gdy kod gwarantuje, że do bufora nie zostanie zapisane więcej bajtów niż wynosi jego rozmiar. Poziom agresywności heurystyki ochrony może być ustawiony przez użytkownika (/GS:1 lub GS:2).

Zabezpieczenia związane ze stack cookies

Przełącznik /GS to jednak nie tylko stack cookies ale też inne mechanizmy zabezpieczające przed efektami przepełnienia bufora na stosie. W Visual Studio 2005 wprowadzono modyfikowanie kolejności zmiennych – bufory umieszczane są pod wyższymi adresami w pamięci (niżej na stosie) aby chronić wskaźniki do funkcji oraz inne zmienne, np. zawierające informacje o uprawnieniach. Ponadto dodano umieszczanie na stosie kopii wrażliwych argumentów (wskaźników, ciągów znaków). Trafiają one na górę stosu, pod niskie adresy i to te kopie są używane potem w kodzie. W ten sposób podczas przepełnienia bufora, gdy zapis występuje na coraz wyższych adresach, nie następuje nadpisanie tych zmiennych ale nadpisywane jest stack cookie. W Visual Studio 2012 dodano sprawdzanie, czy nie następuje zapis znaku NULL do bufora pod indeksem większym lub równym jego rozmiarowi. Dotyczy to buforów, których rozmiar znany jest w czasie kompilacji i miejsc w kodzie, gdzie następuje zapis bajtu o wartości zero (null-character). Funkcja ta została stworzona z myślą o zabezpieczeniu przed nieprawidłową terminacją łańcuchów znaków.

Warto przy okazji wspomnieć, że przed Windows XP SP2 w momencie wykrycia nadpisania cookie, wyjątek był obsługiwany przez procedurę obsługi wyjątków zdefiniowaną w aplikacji. Atakując mechanizm SEH (opisany dalej) można było zamiast tej procedury wykonać shellcode. Lukę tę załatano w XP SP2 – wyjątki generowane przez mechanizm ochrony stosu trafiają do systemowej funkcji, przeznaczonej do obsługi wyjątków nieobsłużonych przez aplikację.

Stack cookies można obejść modyfikując wzorcowe cookie znajdujące się w sekcji .data. Musi być jednak dostępna luka, która umożliwiłaby nadpisanie tego miejsca w pamięci, np. przepełnienie bufora na stercie.

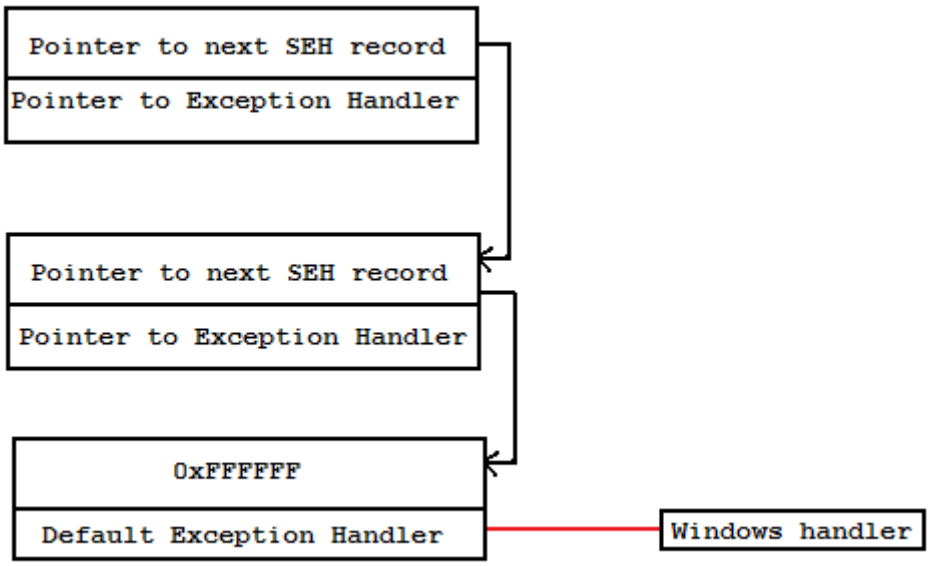
Inną możliwością obejścia stack cookies jest nadpisanie tabeli funkcji wirtualnych obiektu, o ile taki znajduje się na stosie pod adresem wyższym niż adres przepełnianego bufora. Wtedy atakowana aplikacja wywołując metodę obiektu uruchomi shellcode.

Jeśli istnieje możliwość wielokrotnego ponawiania ataku można próbować trafić odpowiednią wartość cookie. Żeby zmniejszyć entropię cookie trzeba mieć informacje, na podstawie których jest ono obliczane. Ogranicza to możliwość ataku do ataków lokalnych.

Structured Exception Handling

Podstawy mechanizmu SEH

Windows oferuje mechanizm obsługi wyjątków znany jako Structured Exception Handling (SEH). Opiera się on na jednokierunkowej liście przechowującej wskaźniki do funkcji obsługujących błędy. W momencie wystąpienia wyjątku, wykonywana jest pierwsza funkcja z listy. Jeśli funkcja nie chce obsłużyć wyjątku (nie obsługuje go wcale lub też konieczne jest jego dodatkowe obsłużenie), zwraca wartość ExceptionContinueSearch. Wtedy brana jest kolejna funkcja z listy. Jeśli funkcja obsługuje wyjątek, zwraca ExceptionContinueExecution. Inne możliwe wartości to ExceptionNestedException oraz ExceptionCollidedUnwind (typ wyliczeniowy EXCEPTION_DISPOSITION). Ostatni element z listy identyfikowany jest po tym, że w swoim wskaźniku na kolejny element pokazuje na adres 0xFFFFFFFF. Z kolei jego wskaźnik na procedurę obsługi wyjątku wskazuje na domyślną systemową funkcję obsługi błędów, która m.in. wyświetla komunikat o wyjątku oraz pozwala zakończyć proces lub go debugować (jeśli w systemie zarejestrowany jest debugger). Tak więc, jeśli wyjątek nie zostanie obsłużony przez żadną funkcję, zajmuje się nim system operacyjny.



Wspomniana lista składa się ze struktur typu EXCEPTION_REGISTRATION, każdy wątek ma swoją własną listę tych rekordów. Wskaźnik na początek listy dla danego wątku jest przechowywany na początku jego struktury TIB (Thread Information Block), wskazywanym przez rejestr segmentowy FS (czyli pod FS:[0]). Jak budowana jest lista? Rekordy EXCEPTION_REGISTRATION są generowane dla każdego bloku _try{}. Umieszczane są one na stosie i dowiązywane do początku listy. W ten sposób rekord najbardziej zagnieżdżonego bloku znajdzie się na początku listy i procedura na którą on wskazuje, będzie wykonana jako pierwsza. Rekordy bloków _try{} danej funkcji będą się znajdować w jednej ramce stosu, należącej do tej funkcji. Cała lista jednak będzie się ciągnąć przez kolejne ramki stosu, analogicznie do ścieżki wywoływania funkcji (call stack).

Co ciekawe, w momencie wystąpienia wyjątku system przechodzi przez listę dwa razy. Za drugim razem do funkcji obsługi wyjątku przekazywana jest flaga EH_UNWINDING, która oznacza, że rozpoczęła się procedura sprzątania po wyjątku, np. usuwania zmiennych automatycznych ze stosu, co wiąże się m.in. z uruchamianiem ich destruktorów, jeśli były to obiekty C++. Wywoływanie procedur obsługi wyjątku aby posprzątały odbywa się od początku listy do tej funkcji, która zgodziła się obsłużyć wyjątek. Procedury wskazywane przez kolejne elementy z listy nie są już wołane. Sprzątanie wykonuje także system operacyjny, który usuwa struktury EXCEPTION_REGISTRATION ze stosu. Po obsłudze wyjątku wygląda więc on tak, jak przed wejściem do bloku _try{}.

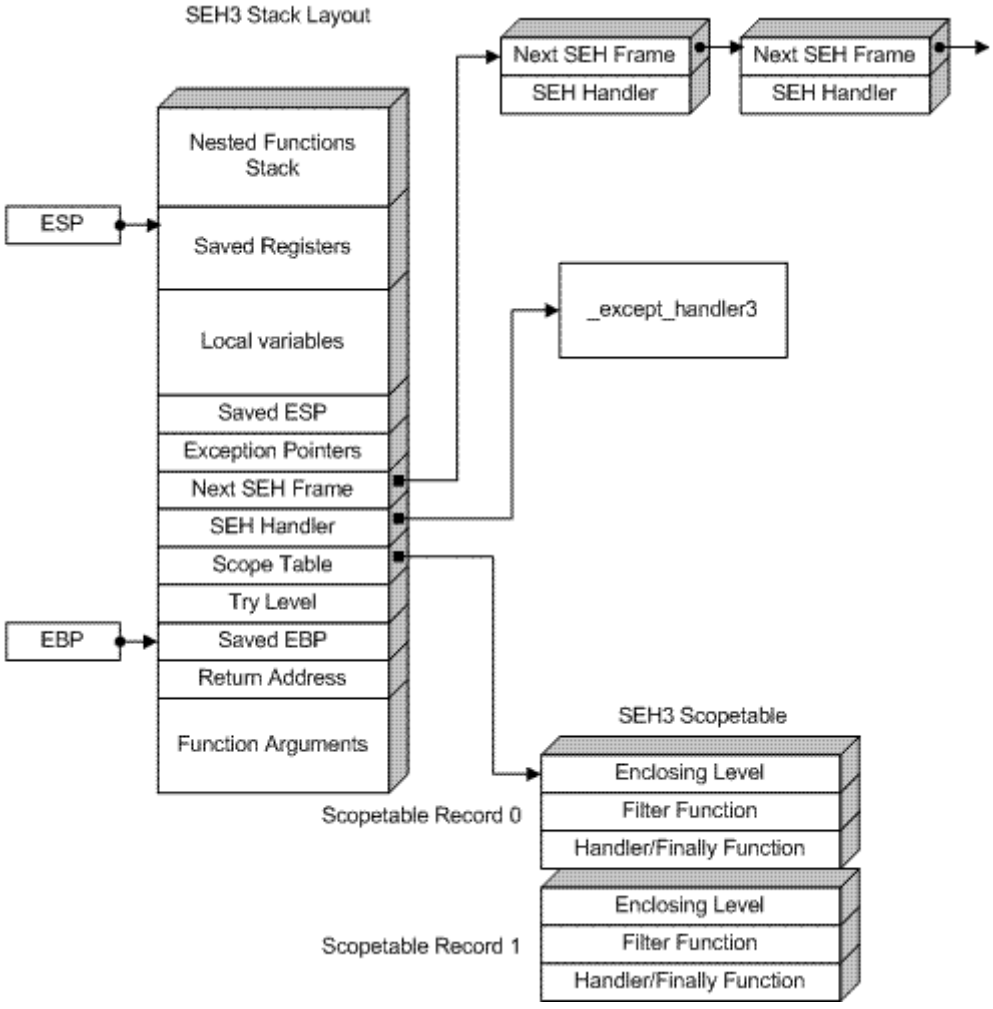
Nakładki na SEH wprowadzane przez kompilatory

Analizując mechanizm SEH trzeba pamiętać, że jest to niskopoziomowa funkcja systemu operacyjnego. Programista praktycznie nigdy się z nim bezpośrednio nie styka. Dzieje się tak, ponieważ kompilatory wprowadzają własną warstwę obsługi wyjątków, która bazuje na mechanizmie systemowym. Należy o tym pamiętać analizując podatności aplikacji skompilowanych np. za pomocą Visual C++.

W aplikacjach skompilowanych narzędziem Microsoftu jest tylko jeden rekord EXCEPTION_REGISTRATION, niezależnie od ilości bloków __try{}. Jest też w związku z tym tylko jedna (z punktu widzenia systemu) funkcja obsługująca wyjątki: __except_handler3 (__except_handler4 w

Visual Studio 2005 i nowszych). W zamian rozszerzona została struktura EXCEPTION_REGISTRATION, zwana w zależności od wersji VS EH3_EXCEPTION_REGISTRATION lub EH4_EXCEPTION_REGISTRATION. Zawiera ona m.in. wskaźnik na tablicę zakresu, która przechowuje struktury SCOPETABLE_ENTRY (<VS2005) lub EH4_SCOPETABLE_RECORD (>=VS2005). W strukturach tych umieszczone są wskaźniki na właściwe funkcje obsługi wyjątków a także wskaźniki na funkcje filtrujące, które decydują czy wywołać daną funkcję obsługi na podstawie określonych warunków. Warto zauważyć, że funkcje filtrujące zwracają wartości takie jak EXCEPTION_CONTINUE_EXECUTION, EXCEPTION_CONTINUE_SEARCH i EXCEPTION_EXECUTE_HANDLER, które wyglądają podobnie do wartości niskopoziomowych (np. ExceptionContinueExecution), ale mają trochę inne znaczenie i inne wartości liczbowe.

Na systemach 64-bitowych SEH zaimplementowane jest inaczej niż na systemach 32-bitowych i struktury EXCEPTION_REGISTRATION nie są przechowywane na stosie, ale w sekcji .pdata, nie są więc podatne na przepełnienia buforów na stosie.



Wykorzystanie SEH do ataku na aplikację

Dlaczego mechanizm SEH interesuje nas z punktu widzenia bezpieczeństwa? Rekordy EXCEPTION_REGISTRATION znajdują się na stosie, podatne są więc na nadpisanie przy przepełnieniu sąsiadującego bufora. Sytuacja nie jest jednak taka sama jak przy nadpisaniu adresu powrotu – ponieważ wyjątek następuje siłą rzeczy przed zakończeniem funkcji, w której wystąpił, manipulując

rekordami SEH możemy obejść zabezpieczenia oparte na stack cookies. Ataki na SEH stały się sposobem radzenia sobie z zabezpieczeniami mającymi chronić przed klasycznymi atakami wykorzystującymi przepełnienie bufora i nadpisanie adresu powrotu. Oczywiście sprawa nie jest tak prosta: zamiast spokojnie czekać na zakończenie funkcji trzeba doprowadzić do wystąpienia wyjątku. Jak też zwykle bywa w takich przypadkach, pojawiły się też zabezpieczenia przeciwko atakom na SEH. Warto się więc przyjrzeć, jak można wykorzystać SEH i jakie można napotkać utrudnienia.

Atak z wykorzystaniem SEH opiera się na fakcie, że w strukturze EXCEPTION_REGISTRATION znajduje się wskaźnik na procedurę obsługi wyjątku, który można nadpisać i spowodować wykonanie kodu dostarczonego przez atakującego. Kod ten może znajdować się w buforze, który jest przepełniany, lub też na stercie, dostarczony np. przez heap spraying. W każdym razie, podobnie jak w przypadku klasycznego ataku z przepełnieniem bufora, niewykorzystującym SEH, nie następuje bezpośrednie nadpisanie adresu procedury obsługi wyjątku adresem shellcode'u. Zamiast tego wykorzystuje się adres sekwencji instrukcji, która pozwoli odpowiednio przygotować stos i wykonać skok do shellcode'u. Zwiększa to niezawodność exploita oraz pozwala obejść niektóre zabezpieczenia.

W celu wykonania skoku do shellcode'u trzeba najpierw zdobyć jego adres. Można by w tym celu wykorzystać jeden z rejestrów, w którym zależnie od logiki programu, mógłby zawierać adres bufora z shellcodem. Jednym jednak z zabezpieczeń wprowadzonych przez Microsoft w Windows XP SP1 jest zerowanie rejestrów (z wyjątkiem oczywiście ESP). Trzeba więc szukać adresu gdzie indziej, np. na stosie. Okazuje się, że z pomocą przychodzi tu fakt, że wywołując procedurę obsługi wyjątku Windows odkłada dla niej na stosie jej argumenty. Drugim argumentem jest EstablisherFrame, wskaźnik na rekord EXCEPTION_REGISTRATION. Nie jest to więc co prawda wskaźnik na bufor z shellcodem, ale jest to miejsce znajdujące się w znanej odległości od bufora. Dodatkowo jest to miejsce w pamięci, którego zawartość jest kontrolowana przez atakującego, który przepełniając bufor nadpisuje wspomniany rekord. W tej sytuacji możliwe jest stworzenie exploitu, który wykona kod dostarczony przez atakującego.

Jak taki exploit działa? Przepełniając bufor nadpisuje rekord EXCEPTION_REGISTRATION, umieszczając, w polu wskaźnika na procedurę obsługi wyjątku, adres sekwencji opcode'ów, która pozwoli na skok do adresu znajdującego się w trzecim elemencie od góry stosu. Sekwencji tej trzeba poszukać w kodzie załadowanych przez aplikację modułów. Dlaczego trzeci element jest interesujący? Na stosie są w bowiem w tym momencie odłożone argumenty dla procedury obsługi wyjątku. Ponieważ odkładane są one w odwrotnej kolejności, argument drugi, czyli EstablisherFrame, umieszczony jest na trzecim miejscu od góry stosu. Powszechnie używaną sekwencją instrukcji jest POP, POP, RET. Zdejmie ona ze stosu czwarty i trzeci parametr a następnie wykona skok pod adres zapisany w parametrze drugim. Procesor skoczy więc do rekordu EXCEPTION_REGISTRATION, a konkretnie do pola przechowującego wskaźnik do następnego rekordu z listy. Pole to również jest kontrolowane przez atakującego i zostało ono uprzednio nadpisane instrukcją skoku do shellcode'u. Jeśli jest on w buforze, który został przepełniony, wystarczy skok o znany offset. Jeśli natomiast został umieszczony na stercie, wówczas zamiast adresu sekwencji POP, POP, RET można od razu podać potencjalny adres shellcode'u. Żeby jednak mieć szansę trafienia w odpowiedni adres, trzeba najpierw wykonać heap spraying.

Zabezpieczenia przed atakami na SEH

Ponieważ ataki z wykorzystaniem Structured Exception Handling stały się popularną metodą obchodzenia stack cookies, Microsoft postanowił je ograniczyć wprowadzając w Windows XP SP2 zabezpieczenie SafeSEH. Powoduje ono, że sprawdzanych jest kilka warunków mających ograniczyć możliwość napisania działającego exploitu:

- rekord EXCEPTION_REGISTRATION musi być na stosie
- rekord musi wskazywać na procedurę, która nie znajduje się na stosie
- jeśli moduł ma włączone zabezpieczenie SafeSEH, procedura wskazywana przez rekord musi być na liście procedur zapisanych w nagłówku modułu
- jeśli jest włączony DEP:
    - procedura musi być w obrazie modułu
    - procedura nie może leżeć w stronie pamięci, która nie jest wykonywalna

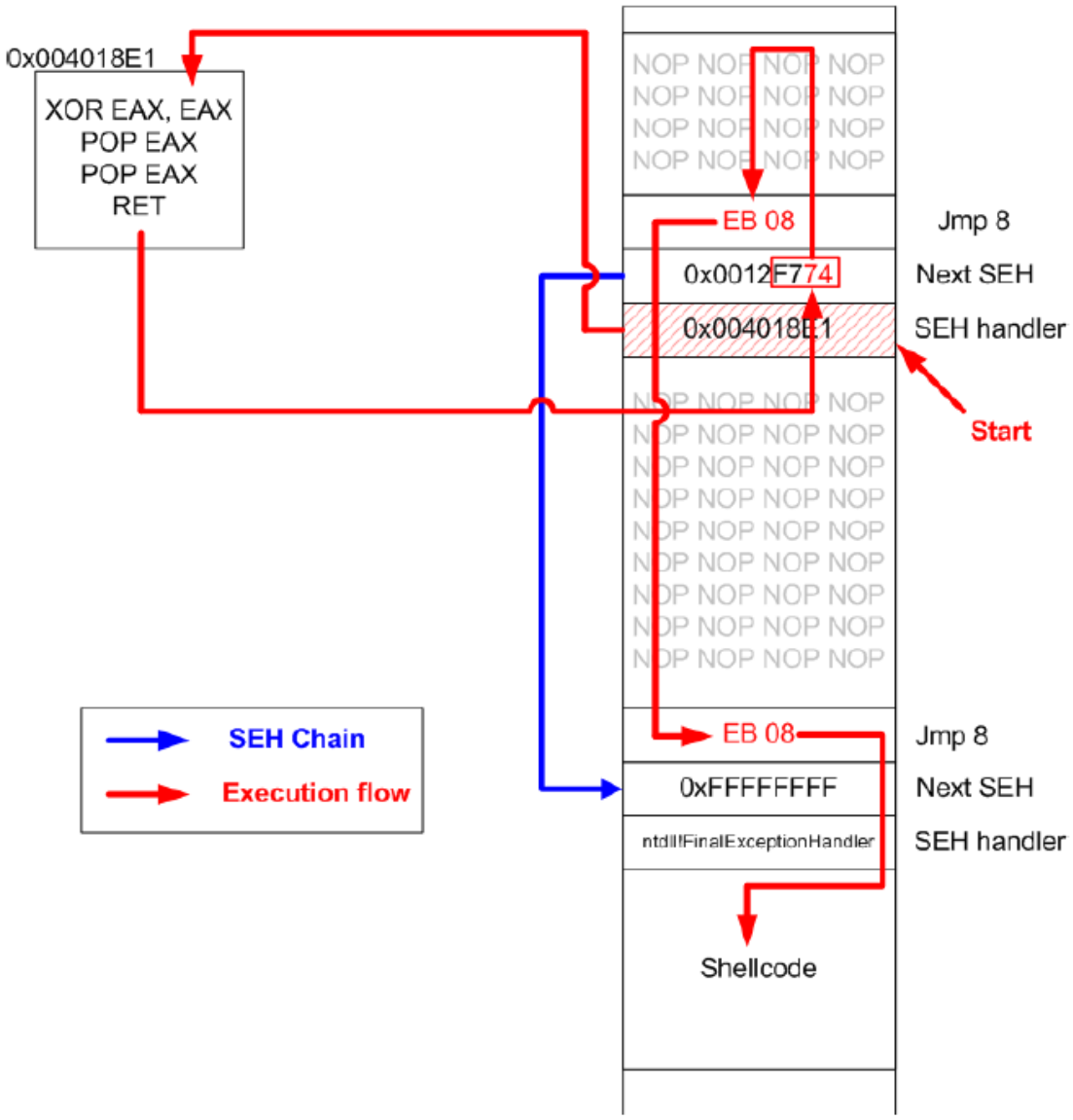
Choć SafeSEH jest zabezpieczeniem realizowanym przez system operacyjny, włączane jest na etapie kompilacji dla poszczególnych modułów. Dlatego podstawową metodą obejścia SafeSEH jest wyszukanie wśród ładowanych przez aplikację modułów takich, które nie mają włączonego SafeSEH. Można potem w nich wyszukać sekwencje opcode'ów POP, POP, RET (lub analogicznych). SafeSEH nie zabezpiecza też przed atakami, w których jako procedura obsługi wyjątku podany zostanie adres ze sterty.

Ponieważ atakujący poradzili sobie z SafeSEH, Microsoft dodał w Windows Vista kolejne zabezpieczenie: Structured Exception Handling Overwrite Protection (SEHOP). Wprowadza ono walidację listy rekordów EXCEPTION_REGISTRATION. Nadpisanie rekordu przez przepełnienie bufora na stosie niszczy jej ciągłość i jest wyraźnym znakiem ataku. Walidacja polega na przejściu listy i sprawdzeniu, czy rozpoczynając od jej pierwszego elementu można osiągnąć element ostatni, w którym wskaźnikiem na kolejny element jest wartość 0xFFFFFFFF, a adresem procedury obsługi wyjątku jest adres funkcji FinalExceptionHandler() z modułu ntdll.dll.

Obchodzenie zabezpieczenia SEHOP

Jednym ze sposobów obejścia SEHOP jest taka modyfikacja wskaźnika na następny rekord EXCEPTION_REGISTRATION, aby zawierał on opcode skoku do shellcode'u, ale jednocześnie był prawidłowym wskaźnikiem na kolejny prawidłowy rekord i nie zaburzał ciągłości listy. Ten nowy rekord atakujący musi umieścić na stosie przy nadpisywaniu bufora. Tworzenie wskaźnika, który jednocześnie pokazywałby na nowy rekord a jednocześnie mógł zostać zinterpretowany jako instrukcja skoku do shellcode'u nie jest proste. Wśród instrukcji skoku tylko jedna posiada opcode, który mógłby być częścią adresu. Dlaczego? Adres musi być wyrównany do 4 a spełnienie tego warunku zapewnia tylko opcode 0x74, należący do instrukcji JE. Jest to jednak skok warunkowy, wykonywany jeśli ustawiona jest flaga Z. Jest ona ustawiana przez instrukcje arytmetyczne, które w wyniku dały zero. W związku tym trzeba taką instrukcję wykonać, dodając ją do sekwencji POP, POP, RET. Oznacza to konieczność znalezienia sekwencji opcode'ów, która będzie zawierać opcode instrukcji zwracającej zero. Może to być np. XOR, POP, POP, RET, gdzie XOR działa na tym samym rejestrze, np. XOR EAX, EAX. Taka sekwencja jest dosyć częsta w funkcjach zwracających zero, nie jest więc trudno ją znaleźć. Oprócz konieczności ustawienia flagi trzeba też wziąć pod uwagę, że nie można skoczyć o dowolną ilość bajtów, ponieważ offset instrukcji skoku jest jednocześnie częścią wskaźnika. W związku z tym skok następuje w dalsze miejsce. Żeby nie powodowało to problemów,

tę część pamięci wypełnia się opcode'ami instrukcji NOP (0x90 dla x86). Między instrukcjami NOP a rekordem EXCEPTION_REGISTRATION umieszcza się instrukcję skoku o 8 bajtów aby przeskoczyć rekord.



Inne podejście do pokonania zabezpieczenia SEHOP to pozostawienie rekordu EXCEPTION_REGISTRATION w stanie nienaruszonym, ale nadpisanie wskaźnika na tabelę zakresu oraz jej indeksu. W ten sposób oryginalna procedura _except_handler3 jest wykorzystywana do wykonania shellcode'u. Ponieważ w aplikacjach skompilowanych za pomocą VS2005 lub nowszych wykorzystywana jest funkcja _except_handler4, która sprawdza między innymi wartości cookie na stosie, można zmodyfikować wskaźnik aby pokazywał jednak na _except_handler3, jeśli jest dostępny w module albo podobną funkcję.

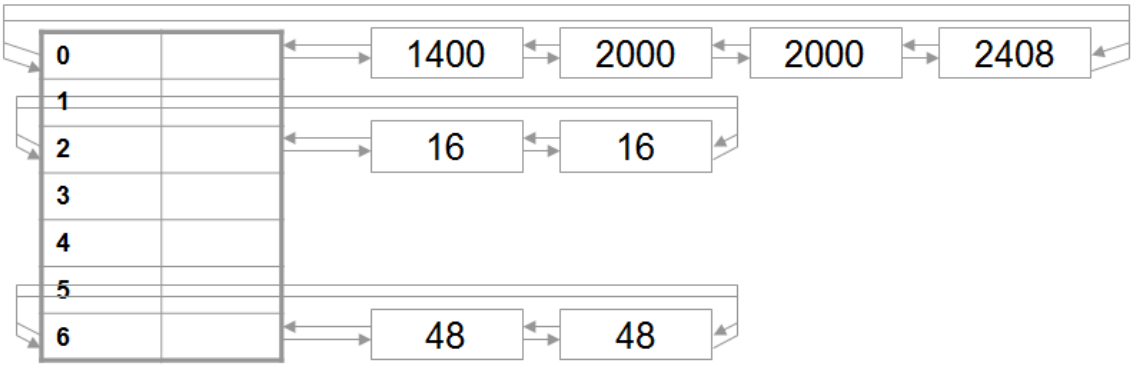
Przepełnienie na stercie

Przepełnienia na stercie bywają traktowane jako mniej groźne od tych na stosie. Nie jest to jednak prawdą i również przepełnienia na stercie pozwalają przeprowadzać skuteczne ataki na podatne aplikacje.

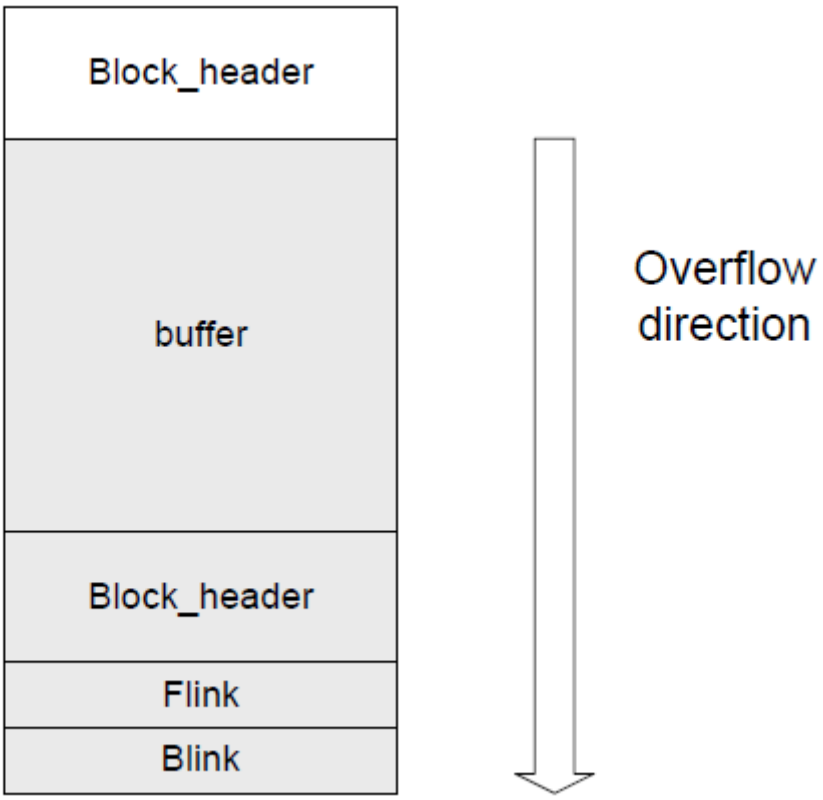
W przypadku przepełnień na stosie wykorzystywany był fakt, że były tam przechowywane dane wpływające na ścieżkę wykonania programu, jak rekord SEH czy adres powrotu. W przypadku sterty sytuacja jest nieco inna, nadpisać można dane używane do zarządzania stertą, co można wykorzystać do nadpisania dowolnego miejsca w pamięci.

Podstawowy sposób ataku z wykorzystaniem struktur sterty

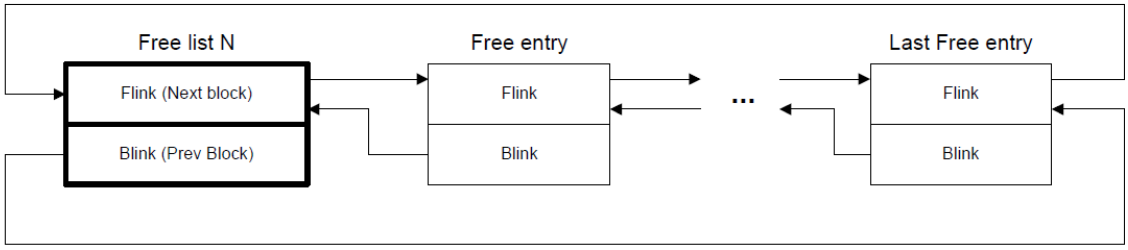
W Windows XP pod offsetem 0x178 względem początku sterty znajduje się tablica FreeList zawierająca 128 struktur typu LIST_ENTRY, które zawierają wskaźniki do dwukierunkowych list wolnych, nieprzydzielonych obszarów. Jest to jedna z ważniejszych struktur wykorzystywanych przez Windows do zarządzania pamięcią. Proces zarządzania stertą jest dosyć złożony i wykorzystywane są też inne struktury pomocnicze, nie są one jednak tutaj omawiane. Pierwszy element tablicy, FreeList[0], wskazuje na listę obszarów o różnym rozmiarze, lecz większym lub równym 1kB a mniejszym od 512 kB. Drugi element listy jest nieużywany. Natomiast kolejne elementy listy FreeList zawierają wskaźniki na listy obszarów o rozmiarze równym indeksowi elementu pomnożonemu przez 8. Przykładowo FreeList[6] wskazuje na listę obszarów o rozmiarze 48 bajtów. Wynika to z tego, że alokacje dotyczą bloków o rozmiarze 8 bajtów. Ponieważ listy wskazywane przez tablicę FreeList są dwukierunkowe, zawierają w swoich elementach wskaźniki na poprzedni i następny element. Podczas alokacji bloku, element który go zawiera jest usuwany z listy, co wiąże się z koniecznością aktualizacji wskaźników w bloku poprzednim oraz następnym aby zachować ciągłość listy. Ma to daleko idące konsekwencje w momencie wystąpienia przepełnienia.



Jeśli zaalokowanym bloku istnieje bufor, a blok ten sąsiaduje z blokiem niezaalokowanym, to przepełnienie bufora spowoduje nadpisanie nagłówka bloku niezaalokowanego, w tym wskaźników.



W momencie, gdy aplikacja będzie chciała zaalokować nadpisany blok, dojdzie do operacji na nadpisanych wskaźnikach, znajdujących się w polach Flink i Blink. Blok, który wskazywał na alokowany blok, musi wskazywać na blok następny. Konieczne jest więc zaktualizowanie pola Flink wskazywanego przez pole Blink alokowanego bloku polem Flink alokowanego bloku. Windows zapisuje więc wartość z pola Flink pod adres wskazywany przez Blink. Ponieważ atakujący nadpisując wartości Flink i Blink kontroluje ich wartość, może w efekcie nadpisać dowolne miejsce w pamięci dowolną wartością. Musi tylko tak dobrać ilość danych nadpisujących, aby nadpisywany adres wypadł w miejscu pola Blink, a nowa wartość w miejscu pola Flink. Aktualizacja wskaźników w momencie usuwania elementu z listy jest jednak dwuetapowa. Drugi etap to aktualizacja następnego elementu, aby jako element poprzedni miał element, który był dotychczas poprzednim dla elementu usuwanego. Pole Blink elementu wskazywanego przez pole Flink elementu usuwanego musi więc zostać nadpisane wartością pola Blink elementu usuwanego. Tak więc dane dostarczone przez atakującego zostaną użyte do wykonania dwóch zapisów do pamięci, z czego jeden jest zamierzony a drugi jest efektem ubocznym. Atakujący nadpisując dany adres daną wartością spowoduje także nadpisanie adresu równego danej wartości z pierwszego zapisu powiększonej o 4, danym adresem z pierwszego zapisu jako wartością. Bardzo często może to spowodować niepożądany efekt, np. wyjątek dostępu do pamięci. Istnieją jednak metody wykorzystania przepełnienia na stercie, w których to dodatkowe przypisanie nie jest utrudnieniem.



Inne metody ataku

Popularną metodą wykorzystania przepełnienia na stercie jest nadpisanie wskaźnika filtra nieobsłużonych wyjątków (Unhandled Exception Filter), konfigurowanego normalnie za pomocą funkcji SetUnhandledExceptionFilter(). Położenie tego wskaźnika należy sprawdzić analizując tę funkcję, przykładowo pod Windows XP SP1 umieszcza ona wskaźnik UEF pod adresem 0x77ED73B4. Znając adres wskaźnika można nadpisać go adresem shellcode'u pod warunkiem, że jest deterministyczny. W większości jednak przypadków trzeba skorzystać z adresu instrukcji, która spowoduje skok do shellcode'u umieszczonego przez atakującego na stercie. Można tu skorzystać z adresu odpowiedniej sekwencji opcode'ów pochodzącej z jednego z załadowanych modułów, oczywiście nieobjętych ASLR. Jakiej jednak sekwencji szukać? Aby wykonać skok w prawidłowe miejsce, wykorzystać można fakt, że rejestr EDI zawiera adres struktury EXCEPTION_POINTERS. Nie ma w niej co prawda wskaźnika na shellcode, ale 0x78 bajtów dalej znajduje się adres nagłówka nadpisanego bloku. Potrzebna jest więc sekwencja odpowiadająca instrukcji call dword ptr[edi+0x78]. Ponieważ w nagłówku są wskaźniki Flink i Blink, na jego początku musi znaleźć się rozkaz krótkiego skoku, który pozwoli je przeskoczyć i trafić w początek znajdującego się za nimi shellcode'u. Tak więc w momencie wystąpienia wyjątku nastąpi skok do modułu, w którym znajduje się instrukcja skoku do nagłówka nadpisanego bloku. Stamtąd nastąpi skok do shellcode'u.

Przepełniając stertę można wykorzystać także mechanizm SEH omówiony przy okazji przepełnień na stosie. Struktury SEH znajdują się właśnie na stosie, jednak początek łańcucha struktur EXCEPTION_REGISTRATION jest wskazywany przez wskaźnik znajdujący się w TEB (Thread Environment Block). Istnieją jednak oddzielne bloki TEB dla poszczególnych wątków, może więc pojawić się trudność w ustaleniu który blok TEB nadpisać. Jeśli jednak nie jest to problemem, można wykorzystać fakt, że TEB pierwszego wątku znajduje się zwykle pod adresem 0x7FFDE000 i ten adres nadpisać. Trzeba jednak pamiętać, że mechanizm ASLR może zmienić położenie TEB na losowe.

W Windows XP wprowadzono wektorową obsługę błędów (Vectored Exception Handling). Wykorzystuje ona struktury _VECTORED_EXCEPTION_NODE, podobne do struktur EXCEPTION_REGISTRATION, ale umieszczone są one na stercie, nie na stosie. Również tworzą listę, ale dwukierunkową. W Windows XP SP1 wskaźnik na początek listy znajduje się pod adresem 0x77FC3210. Wywołując przepełnienie bloku na stercie można go nadpisać wskaźnikiem do shellcode'u. Nie ma tu niestety uniwersalnego sposobu. Można np. poszukać wskaźnika do zaalokowanego bloku na stosie, zapewne jest przechowywany w zmiennej lokalnej albo był przekazywany do funkcji a w tych wypadkach wartość zapisywana jest na stosie. Ponieważ wskaźnik na procedurę obsługi wyjątku jest trzecim polem w strukturze _VECTORED_EXCEPTION_NODE,

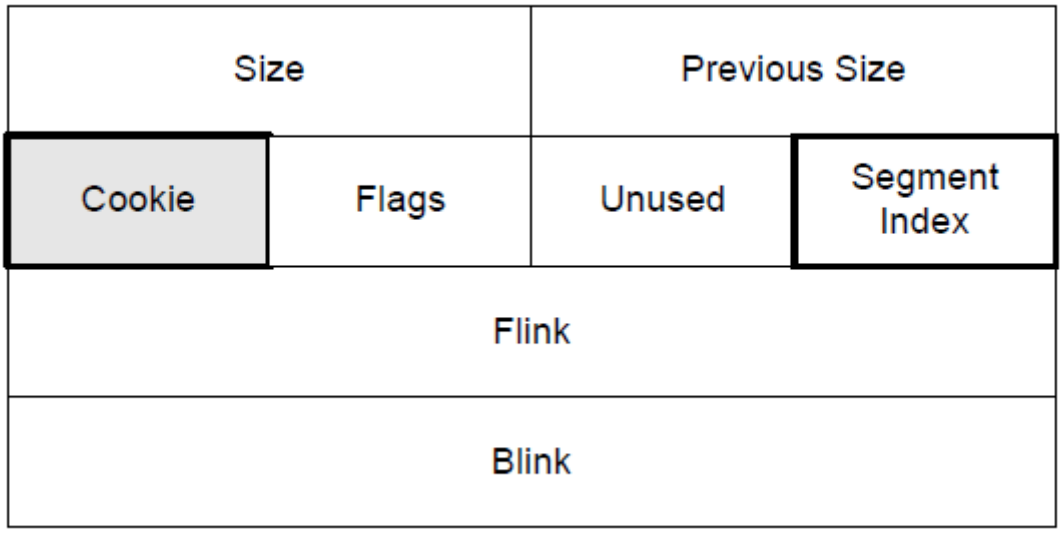
należy pamiętać aby zamiast bezpośredniego wskaźnika na shellcode użyć adresu pomniejszonego o 8.

W systemach Windows NT – XP można było wykorzystać strukturę TEB w jeszcze inny sposób. Znajdował się tam m.in. wskaźnik na funkcję RtlEnterCriticalSection(), był on pod stałym adresem 0x7FFDF020. Z tego wskaźnika korzystała funkcja RtlAccquirePebLock() wołana przez ExitProcess(). Za pomocą przepełnienia na stercie można było nadpisać ten wskaźnik i spowodować wykonanie shellcode'u gdy wywoływana jest funkcja ExitProcess(), np. przy wystąpieniu wyjątku kończącego działanie aplikacji.

W przypadku ataków wykonujących przepełnienie bloku na stercie jedną z pierwszych czynności wykonywanych przez shellcode powinno być naprawienie struktury sterty. W przeciwnym razie funkcje wywoływane przez shellcode, które korzystają ze sterty, mogą doprowadzić do naruszenia ochrony pamięci. Najprostszą metodą, łatwiejszą niż odtwarzanie stanu sterty sprzed ataku, jest nadanie jej wyglądu nowej, świeżo utworzonej sterty.

Zabezpieczenia chroniące stertę

W systemie Windows XP SP2 wprowadzono dwa zabezpieczenia mające utrudnić ataki z wykorzystaniem przepełnień na stercie. Jedno z nich to cookies podobne do tych zabezpieczających stos. W XP SP2 zmienił się nagłówek bloku, indeks segmentu przesunięty na miejsce indeksu tagu, a na jego miejsce trafiło cookie. Jest to więc losowy bajt znajdujący się pod offsetem 0x05 od początku nagłówka. Wartość cookie sprawdzana jest podczas alokacji bloku, gdy jest usuwany z listy wolnych bloków. W przypadku wykrycia niezgodności generowany jest wyjątek. Drugie z dodanych w SP2 zabezpieczeń to tzw. safe unlinking, które jest po prostu weryfikacją wskaźników w alokowanym bloku.



Aby obejść te zabezpieczenia, można wykorzystać tzw. lookaside list. Jest to dodatkowa lista wolnych bloków, do której trafiają bloki, które były już alokowane. Jej zadaniem jest przyspieszenie działania mechanizmu zarządzania stertą. Nie jest objęta zabezpieczeniami. Aby wykorzystać ten fakt, trzeba zaalokować a następnie zwolnić blok o rozmiarze nieprzekraczającym 1016 bajtów. Dzięki temu po

zwolnieniu blok ten trafi do lookaside list. Jeśli atakującemu uda się dokonać przepełnienia w bloku poprzednim, nadpisze on pole Flink bloku znajdującego się na liście. Jeśli teraz zostanie wykonania alokacja o rozmiarze równym zwalnianemu blokowi, to zostanie zaalokowany ten właśnie blok z listy lookaside a jego pole Flink trafi do listy jako głowa. Wartość ta będzie traktowana jako adres wolnego bloku. Jeśli teraz nastąpi alokacja, do aplikacji zostanie zwrócona ta właśnie wartość. W ten sposób nastąpi zapis pod adres dostarczony przez atakującego. Jeśli dane użyte do tego zapisu będą także kontrolowane przez atakującego, otrzymujemy w efekcie możliwość zapisu do 1016 bajtów pod dowolny adres w pamięci. Wygląda to bardzo atrakcyjnie, lecz jak widać, konieczne jest aby atakowana aplikacja wykonała ściśle określoną sekwencję alokacji i dealokacji. Warunek ten może być trudny do spełnienia, np. jeśli aplikacja nie oferuje obsługi języka skryptowego, co np. w przypadku przeglądarek internetowych nie jest problemem. W aplikacji musi być też oczywiście używana lista lookaside.

W Windows Vista wprowadzono dalsze zabezpieczenia mające chronić stertę. Zrezygnowało z listy FreeList a listę lookaside zastąpiono mechanizmem Low Fragmentation Heap (LFH). Dodano szyfrowanie nagłówków bloków poprzez XORowanie z losową wartością. Cookie sprawdzane jest częściej i używane do sprawdzania integralności innych pól. Adres bazowy sterty jest losowy, mechanizm ASLR zapewnia entropię na poziomie 5 bitów. Szyfrowane są także wskaźniki do funkcji, które są przechowywane na stosie. Wprowadzono możliwość natychmiastowego zakańczania procesu w momencie wykrycia modyfikacji sterty. Ponadto zmodyfikowano algorytmy alokacji aby działały mniej deterministycznie z punktu widzenia atakującego. W Windows 8 wprowadzono losowość kolejności alokacji bloków. Atakujący nie może już zakładać, że kolejny blok zostanie zaalokowany po poprzednim.

Oczywiście ataki z wykorzystaniem przepełnień na stercie są nadal możliwe. Są jednak trudniejsze i atakujący musi nadpisywać inne struktury niż w czasach ataków na Windows XP, np. wykorzystywane przez LFH.

Data Execution Prevention

DEP ma za zadanie zabezpieczyć przed uruchomieniem kodu zawartego w payloadzie dostarczonym przez exploit. Opiera się na oznaczaniu stron w pamięci jako miejsc, z których kod nie może być wykonany. W przypadku próby wykonania kodu ze strony, która ma wyłączoną wykonywalność, generowany jest wyjątek STATUS_ACCESS_VIOLATION. DEP wykorzystuje sprzętowe wsparcie ze strony procesora. Dla procesorów AMD jest to bit NX (Never eXecute), dla Intela jest to bit XD (eXecute Disable), a w rodzinie ARM używana jest nazwa XN (eXecute Never). Na systemach 32-bitowych DEP działa dzięki PAE, na systemach 64-bitowych jest obsługiwany natywnie. Istnieją też programowe implementacje DEP, takie jak np. W^X czy Exec Shield. O software DEP mówi też Microsoft, jednak odnosi się to tak naprawdę do SafeSEH, konkretnie do sprawdzania, czy procedura obsługi wyjątku znajduje się w wykonywalnej stronie pamięci.

Pod Windows DEP konfigurowany jest z poziomu systemu, aplikacji oraz kompilatora. W przypadku kompilatora Visual C++ DEP włącza się za pomocą przełącznika /NXCOMPAT, który informuje system, czy dana aplikacja jest kompatybilna z DEP. Jeśli jest, Windows może dla niej włączyć zabezpieczenie DEP. Użytkownik może skonfigurować DEP w systemie na cztery sposoby: optin, optout, alwayson, alwaysoff. Ustawienie optin oznacza, że DEP włączony jest dla aplikacji i usług systemowych,

pozostałe programy użytkownik musi ręcznie dodać do listy chronionych. W przypadku optout domyślnie chronione są wszystkie aplikacje z wyjątkiem wskazanych przez użytkownika. Alwayson włącza DEP dla wszystkich programów bez wyjątku a alwaysoff wyłącza dla wszystkich bez wyjątku. Konfiguracja DEP dla danego uruchomionego procesu może być wykonywana za pomocą funkcji NtSetInformationProcess() oraz SetProcessDEPPolicy(), z których zalecana jest ta druga. Jeśli aplikacja potrzebuje zaalokować pamięć aby wykonać w niej kod, może skorzystać z funkcji VirtualAlloc[Ex]. Z kolei do zmiany wykonywalności już użytej pamięci służy funkcja VirtualProtect[Ex].

Co ciekawe funkcje te pozwalały konfigurować ustawienia wykonywalności zanim powstał DEP. Do tego linkery ustawiały flagi w nagłówku PE prawidłowo a programiści prawidłowo ustawiali flagi w wywołaniach API, ponieważ były udokumentowane jako działające. Dlatego też po wprowadzeniu obsługi DEP w Windows istniejące oprogramowanie było zgodne z nowym zabezpieczeniem bardziej niż można się było tego spodziewać. Niekompatybilność dotyczyła m.in. aplikacji korzystających z bibliotek ATL, które wykonywały małe fragmenty kodu na stercie. Microsoft wprowadził w Windows specjalne obejście tego problemu.

DEP ma za zadanie zabezpieczać przed wykonaniem kodu, który nie leży w obszarze wykonywalnym. W związku z tym pomysł na obejście DEP to wykorzystanie kodu, który już znajduje się w obszarze wykonywalnym będąc częścią atakowanej aplikacji. Oczywiście atakowana aplikacja nie zawiera shellcode'u. Zawiera jednak dużą liczbę małych fragmentów kodu, które mogą być przydatne atakującemu. Można je wykorzystać na kilka sposobów:

- zrealizować za ich pomocą funkcję shellcode'u
- włączyć wykonywalność dla strony w której umieszczony jest shellcode i skoczyć do niego
- zaalokować wykonywalny obszar, skopiować do niego shellcode i uruchomić go
- wyłączyć DEP dla procesu (jeśli nie ma włączonej funkcji permanent DEP)

Niezależnie od tego, który ze sposobów zostanie wybrany, stosuje się tę samą technikę polegającą na skoku do małych fragmentów kodu realizujących konkretne funkcje. Aby móc wykonać kilka takich fragmentów, szuka się takich, które kończą się instrukcją RET. W ten sposób wykonać sekwencję takich fragmentów przez umieszczenie ich adresów na stosie. Muszą się tam też znaleźć argumenty dla wywoływanych funkcji. Na stosie atakujący musi więc stworzyć jakby historię wywołań, a zaatakowana aplikacja wykonując wspomniane fragmenty kodu de facto ciągle powraca. Stąd też technika ta nazywana jest Return Oriented Programming. Duża część z używanych fragmentów kodu funkcje libc lub WinAPI, dlatego popularne jest również określenie return-to-libc (ret2libc).

Address Space Layout Randomization

ASLR jest zabezpieczeniem, które Microsoft wprowadził w Windows Vista. Polega ono na tym, że różne obiekty należące do procesu umieszczane są pod losowymi adresami. Utrudnia to znacznie pisanie exploitów, które zwykle wykorzystują określone, predefiniowane adresy, np. w celu wywoływania funkcji WinAPI. ASLR powoduje, że losowe stają się adresy stert i stosów procesu, adres bazowy procesu, adresy bazowe bibliotek DLL oraz obszary PEB (Process Environment Block) i TEB (Thread Environment Block).

O włączeniu losowości adresu bazowego dla pliku EXE lub DLL decyduje pole DllCharacteristics w nagłówku PE pliku wykonywalnego. Jeśli jest w nim ustawiony bit IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE, wówczas plik wykonywalny jest oznaczony jako kompatybilny z ASLR. W Visual C++ bit ten jest ustawiany za pomocą przełącznika /DYNAMICBASE w ustawieniach linkera. Funkcję ASLR można było kontrolować w Windows Vista przez wpis w rejestrze, w Windows 7 nie można jej wyłączyć ani wymusić włączenia. W przypadku plików EXE funkcja ASLR powoduje wprowadzenie losowości adresu bazowego oraz stosu, w przypadku bibliotek DLL losowy jest adres bazowy. Adres sterty jest losowy zawsze, niezależnie od ustawień ASLR. Ze względu na to, że biblioteki DLL są współdzielone przez wiele procesów, ich adresy zmieniają się po każdym uruchomieniu systemu, a nie przy uruchamianiu procesu. W trakcie działania sesji systemu operacyjnego są więc stałe.

Ponieważ ASLR jest włączane dla poszczególnych modułów, częsta może być sytuacja, że nie wszystkie biblioteki DLL załadowane przez dany proces są objęte ASLR. W ten sposób część kodu może nadal znajdować się pod przewidywalnymi adresami i być dostępna dla atakującego, np. przy atakach wykorzystujących ROP.

Losowość wprowadzana przez ASLR utrudnia znalezienie różnych obiektów w pamięci ale nie uniemożliwia. Często wykorzystywaną techniką, szczególnie w atakach na przeglądarki, jest tzw. heap spraying. Polega on na zaalokowaniu dużych obszarów pamięci na stercie i nadpisaniu ich odpowiednimi danymi. W ten sposób można z dużym prawdopodobieństwem trafić na miejsce, do

którego odwołuje się atakowana aplikacja. Zapisywane dane to często małe fragmenty shellcode'u rozdzielone dużymi obszarami wypełnionymi instrukcjami NOP. Dzięki temu można wykonać skok nie znając dokładnego położenia shellcode'u. Po skoku nastąpi „ześlizgnięcie" się po NOPach do shellcode', stąd nazwa „NOP slide"/"NOP sled". Takie przygotowanie danych było już wspomniane przy omawianiu obejścia zabezpieczenia SEHOP.

Inną ciekawą metodą obejścia ASLR jest wykorzystanie faktu, że nie cały adres jest losowy, zwykle losowość dotyczy tylko dwóch najstarszych bajtów. Wykonując więc atak poprzez przepełnienie bufora można nadpisać tylko młodsze bajty. W ten sposób co prawda nie można dowolnie zmodyfikować adresu, można jednak uniezależnić się od losowości wprowadzanej przez ASLR i np. wskaźnik pokazujący na funkcję z modułu przestawić tak, aby pokazywał na inną funkcję z tego samego modułu.

Atakujący może też starać się wykorzystać dane, do których ma dostęp aby odgadnąć położenie innych danych. Np. adres zmiennej statycznej pozwala określić adres bazowy jej modułu.

Przepełnienie całkowite

Typy całkowitoliczbowe w zależności od języka programowania i architektury różnie się zachowują przy próbie zapisu wartości większej niż są w stanie pomieścić. Zmienna może się nasycić i przyjąć najwyższą możliwą wartość, kolejne próby inkrementacji nie będą już powodowały zmiany jej wartości. Może też nastąpić tzw. przekręcenie się lub zawinięcie, objawiające się tym, że zmienna przybierze najniższą możliwą wartość. Może to powodować nieprzewidziane zachowanie aplikacji szczególnie dla zmiennych ze znakiem. Np. zwiększenie o 1 jednobajtowej zmiennej ze znakiem o wartości 127 spowoduje pojawienie się wartości -128. Analogicznie odjęcie 1 od jednobajtowej zmiennej ze znakiem, o wartości -128 spowoduje, że w zmiennej pojawi się liczba 127. Podobnie może się zdarzyć przy innych operacjach arytmetycznych, np. przy mnożeniu. W ten sposób pomnożenie dwóch liczb dodatnich może dać liczbę ujemną. Takie zachowania bywają często nieprzewidziane przez programistę i mogą spowodować niepożądane działanie aplikacji. Dotyczy to szczególnie przypadków, gdy aplikacja odczytuje rozmiar danych z niezaufanego źródła. Atakujący może np. dostarczyć plik, w którego nagłówku ilość danych zostanie opisana liczbą ujemną. Jeśli program nie zwaliduje odpowiednio takiej wartości, może być podatny na przepełnienie bufora.

Poniższy przykład to podatność CVE-2010-3970 z biblioteki shimgvw.dll:

```
mov [ebp+bmi.bmiHeader.biClrUsed], eax
; [ . . . ]
mov ecx, eax
cmp ecx, 100h
jg error
add esi, 28h
lea edi, [ebp+bmi.bmiColors]
rep movsd; move ds:esi -> ds:edi, powtórzone ecx razy
```

Wartość z rejestru EAX używana przy wypełnianiu struktury bmi jako ilość kolorów w bitmapie. Następnie trafia ona do rejestru ECX, gdzie jest porównywana z liczbą 100h czyli 256d. Problem polega na tym, że porównanie zakłada, że liczba jest ze znakiem, czyli może być ujemna. Jeśli więc w

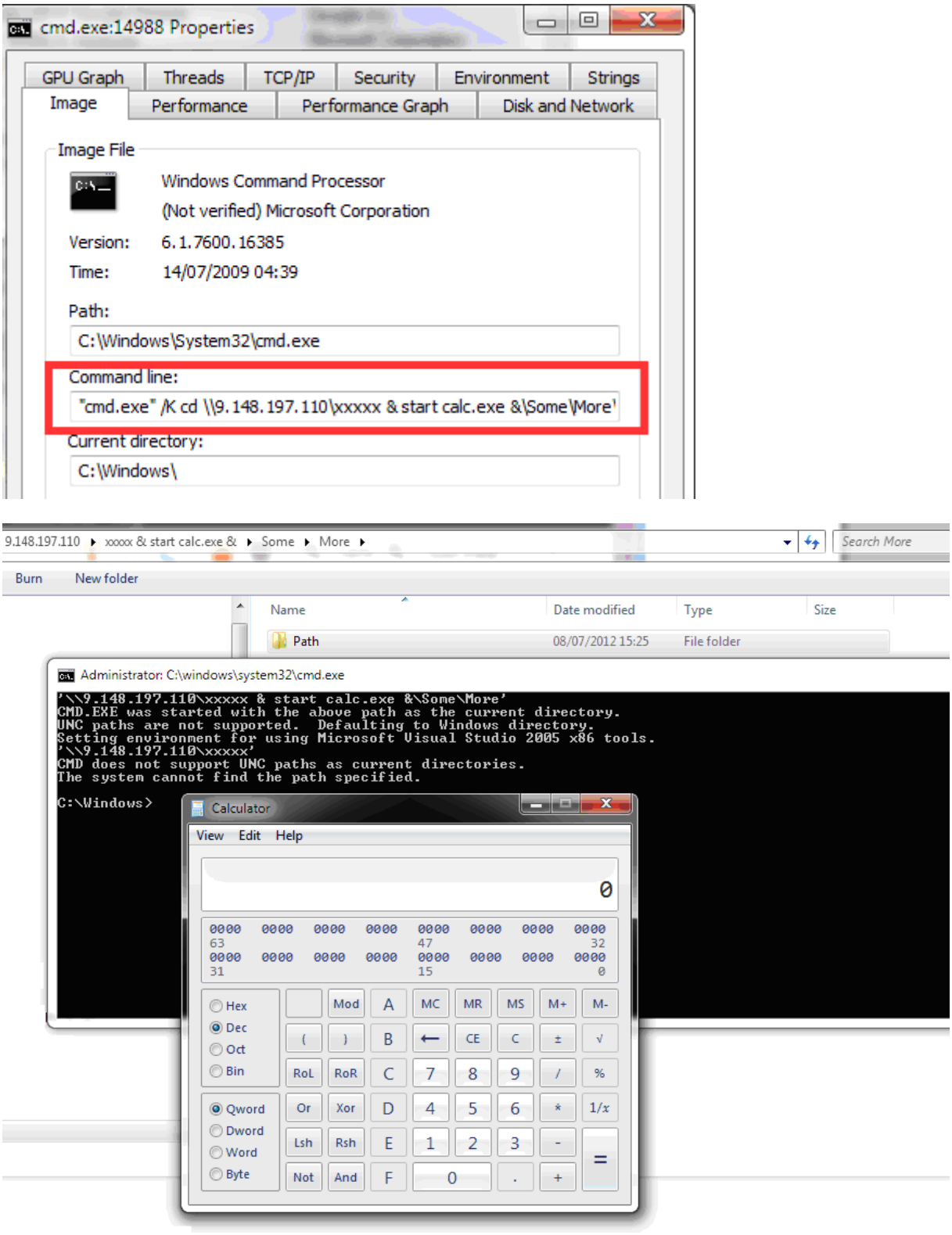
rejestrze ECX będzie znajdować się bardzo duża liczba dodatnia, zostanie ona zinterpretowana jako ujemna, a tym samym mniejsza od 256 (przykładowo dane FFFFFFFFh dla zmiennej ze znakiem zostaną zinterpretowane jako -1 a dla zmiennej bez znaku jako 4294967295). Walidacja przejdzie więc pozytywnie i nastąpi kopiowanie bajtów z buforu wskazywanego przez rejestr ESI do buforu wskazywanego przez rejestr EDI. Instrukcja REP MOVSD będzie próbować przekopiować tyle bajtów ile wynosi wartość rejestru ECX. Ponieważ występuje tam bardzo duża liczba, nastąpi przepełnienie bufora, które w tym konkretnym ataku zostało wykorzystane do nadpisania struktury SEH.

Przechodzenie ścieżki

Wiele aplikacji pracuje ze ścieżkami. Przykładem mogą być serwery WWW, które obsługują żądania pobrania różnych plików z serwera. Serwer musi odpowiednio sparsować ścieżkę nadesłaną przez przeglądarkę aby określić jaki plik ma być pobrany oraz jakie parametry mają być do niego przekazane. Obecnie bowiem bardzo rzadko mamy do czynienia ze statycznymi stronami WWW, obecnie praktycznie wszystkie z nich są skryptami i serwer WWW zwraca nie tyle plik z dysku co wynik działania zapisanego w nim skryptu. Musi przy tym kontrolować, czy przychodzące żądanie HTTP nie dotyczy pliku znajdującego się poza tymi, które tworzą stronę internetową, czy użytkownik ma odpowiednie uprawnienia itd. Ścieżkę muszą też analizować wspomniane skrypty / aplikacje webowe, które otrzymują ścieżki jako parametry. Nieprawidłowa analiza ścieżki powoduje zagrożenie. Atakujący może np. otrzymać dostęp do plików systemu operacyjnego lub przechowujących hasła, może także nadpisać wrażliwe dane. Możliwe jest też takie spreparowanie ścieżki, że atakujący spowoduje wykonanie polecenia na zdalnym systemie.

Szukanie podatności umożliwiającej przechodzenie ścieżki na stronach WWW często rozpoczyna się od sprawdzenia reakcji na ciąg „../”, który oznacza katalog nadrzędny. Niejednokrotnie umożliwia on dostęp do plików znajdujących się poza katalogiem, w którym znajduje się strona internetowa. Ponieważ jest to znany atak i ciąg ten bywa często odfiltrowany, stosuje się jego kodowanie, np. za pomocą UTF-8.

Niepoprawna walidacja ścieżki pozwala też na zdalne wykonanie polecenia. Przykładowo, może istnieć aplikacja, która uruchamia drugą aplikację przekazując do niej argumenty przekazane przez użytkownika. Jeśli wykorzystuje do tego powłokę, atakujący może być w stanie wśród argumentów umieścić polecenie powłoki. Konkretnym przykładem takiej podatności może być MS12-048 (CVE-2012-0175). Dotyczy ona otwierania plików w skojarzony z nimi aplikacjach pod Windows. Przy próbie otwarcia pliku Windows uruchamia skojarzony program podając mu ścieżkę do danego pliku jako parametr. Aby wykorzystać ten mechanizm do wykonania swojego polecenia atakujący musiałby stworzyć plik, którego nazwa mogłaby zostać zinterpretowana jako polecenie powłoki. Ponieważ nazwa pliku jest umieszczana w cudzysłowach, musiałby być w niej cudzysłów lub inny znak oznaczający zakończenie parametru. Powłoka Windows jest jednak zabezpieczona przed takim przypadkiem. Okazało się jednak, że o ile nazwy plików są sprawdzane pod kątem nieprawidłowych znaków, to nie są walidowane nazwy udziałów sieciowych. Atakujący może więc stworzyć udział (np. za pomocą serwera linuksowego), którego nazwa będzie zawierała cudzysłów oraz polecenie.



Użycie po zwolnieniu

Jest to błąd w programie polegający na użyciu obiektu po zwolnieniu zajmowanej przez niego pamięci. Gdy programista zniszczy obiekt, system operacyjny przeznaczy zwolnioną pamięć pod

przyszłe alokacje. W ten sposób zostanie on nadpisany nowymi danymi. Jest to normalne i oczywiste. Problem pojawia się w momencie, gdy aplikacja próbuje odwołać się do zniszczonego obiektu, np. wykonać jego metodę. Ponieważ w tym momencie tablica funkcji wirtualnych obiektu została nadpisana, nastąpi skok pod zupełnie inny adres. Taką sytuację może wykorzystać atakujący dostarczając dane, które spowodują wykonanie jego kodu. Błędy użycia po zwolnieniu są często spotykane w Internet Explorerze, np. MS12-063.

W jaki sposób atakujący może nadpisać zniszczony obiekt? Jeśli aplikacja obsługuje np. skrypty, staje się to bardzo łatwe – atakujący może alokować pamięć i trafić na odpowiedni obszar. Sztandarowym przykładem takiej aplikacji jest przeglądarka internetowa, wiele ataków na przeglądarki to obecnie właśnie ataki wykorzystujące podatność na użycie po zwolnieniu. Atakujący, np. za pomocą JavaScriptu czy też Flasha, albo nawet wczytując duże pliki graficzne, może łatwo zaalokować duże obszary pamięci i umieścić tam swój shellcode. Do zaalokowanej pamięci zapisywane są bajty, które mogą być zinterpretowane jako adres w pamięci, co do którego istnieje duże prawdopodobieństwo, że będzie wskazywać na obszar zaalokowany przez atakującego. W ten sposób jeśli podatna aplikacja odwoła się przez wskaźnik należący do zniszczonego obiektu, odwoła się ostatecznie do tego obszaru. Atakujący, choć może mieć dużą pewność, że jego dane znajdą się w pamięci pod określonym adresem, to nie może być pewien offsetu. Innymi słowy, nie ma żadnej pewności, że pod danym adresem znajdzie się początek zapisywanych przez niego danych. Prawdopodobieństwo to jest bardzo małe. W związku z tym do pamięci są zapisywane serie dużych bloków danych, w których shellcode zajmuje stosunkowo niewielki procent. Pozostała przestrzeń w bloku to bajty będące instrukcjami NOP lub jej odpowiednikami. Jak jednak wspomnieliśmy, atakujący musi nadpisać miejsce pokazywane przez wskaźnik adresem, który mógłby kontrolować. Tak więc wspomniane bajty muszą dać się dekodować jednocześnie jako adres kontrolowany przez atakującego jak i instrukcje NOP lub równoważne. W ten sposób można mieć dużą pewność, że nadpisany zostanie adres pokazywany przez wskaźnik, że adres ten będzie pokazywać na kontrolowany obszar, oraz że skacząc do tego obszaru procesor natrafi na instrukcje NOP a dopiero po nich na shellcode. Ponadto unika się sytuacji, gdy skok następuje w środek shellcode'u. Ponieważ wspomniane serie alokacji wykonywane są na stercie, cała ta technika określana jest mianem heap spraying.

Nieprawidłowa konfiguracja i weryfikowanie uprawnień

Jest oczywiste, że nieprawidłowo skonfigurowane uprawnienia do plików i innych obiektów są luką w bezpieczeństwie. Jednak nie tylko proste błędy konfiguracyjne mogą spowodować, że system lub aplikacja będą podatne na atak. Przykładem może być dziura w jądrze 2.6.39 systemu GNU/Linux. W systemie plików Linuksa dostępne są pliki /proc/<pid>/mem, które dają dostęp do pamięci poszczególnych procesów. Dostęp do nich jest odpowiednio zabezpieczany, jednak w jądrze 2.6.39 pojawiła się luka, która pozwalała na podniesienie uprawnień. Wykorzystywała ona fakt, że można było nakłonić proces do nadpisania swojej własnej pamięci przez podanie mu deskryptora jego pliku mem. Jeśli proces ten miał ustawiony bit SUID, możliwe było wykonanie kodu z uprawnieniami roota.

Ładowanie bibliotek z niezaufanego źródła

Aplikacje dla systemu Windows, chcąc załadować bibliotekę DLL, przeszukują takie katalogi jak katalog własny, katalog systemowy (System32), katalog Windows, katalog bieżący oraz lokalizacje wymienione w zmiennej systemowej PATH. W przypadku katalogu bieżącego staje się to niebezpieczne, gdyż katalog bieżący może być kontrolowany przez atakującego, np. gdy aplikacja pozwala otworzyć dokument. Atakujący może stworzyć udział SMB lub WebDAV a następnie nakłonić ofiarę do otwarcia pliku z tego zasobu. Aplikacja zmieni wtedy katalog bieżący na lokalizację kontrolowaną przez atakującego. Jeśli będzie potem w czasie swego działania próbować otworzyć plik DLL i znajdzie go na udziale atakującego, może zostać wykonany szkodliwy kod przez niego dostarczony. Problem był szczególnie poważny w starszych wersjach Windows, takich jak Windows 2000 SP3 czy Windows XP SP1, tam systemowe funkcje API najpierw przeglądały katalog bieżący, a potem systemowy. Lukę tę załatano, jednak oczywiście łatka nie uniemożliwia programistom pisania aplikacji ładujących biblioteki w niebezpieczny sposób. Kolejnym zabezpieczeniem jaki Microsoft wprowadził, to możliwość wyłączenia ładowania bibliotek z udziałów SMB i WebDAV. Może to zrobić administrator dla całego systemu lub poszczególnych aplikacji za pomocą wpisu w rejestrze. Opisywana dziura jest spotykana także w programach Microsoftu. Od momentu nagłośnienia luki w 2010 roku Microsoft wydał 27 biuletynów bezpieczeństwa zawierających łatki na tę lukę.

Time of check to time of use

Jest to błąd zbliżony do omówionego problemu nieprawidłowej weryfikacji uprawnień, wynikający z możliwości wystąpienia wyścigu. Polega on na tym, że warunek, na którego podstawie aplikacja przyznaje dostęp do określonego zasobu lub nie, może się zmienić pomiędzy sprawdzeniem warunku a użyciem uprawnień. Przykładowo program działającymi z wysokimi uprawnieniami może sprawdzać, czy użytkownik ma uprawnienia do pliku. Jeśli ma, następuje otwarcie pliku do zapisu i zapis danych dostarczonych przez użytkownika. Jeśli atakujący pomiędzy sprawdzeniem uprawnień a otwarciem pliku skasuje go, a w jego miejsce umieści dowiązanie do pliku, do którego nie posiada uprawnień, aplikacja pozwoli mu nadpisać ten plik. Jest to więc podatność pozwalająca na podniesienie uprawnień. Przeprowadzenie takiego ataku wymaga wykonania danych działań w ściśle określonych momentach, gdyż czas pomiędzy obiema operacjami wykonywanymi przez aplikację jest bardzo krótki. Wydawać by się mogło, że trafienie w odpowiedni moment jest bardzo trudne i zagrożenie atakiem niewielkie. Okazuje się jednak, że nie jest to wcale tak skomplikowane i istnieją techniki pozwalające wykonać atak w odpowiedniej chwili. Jednocześnie zabezpieczenie się przed tego typu atakami jest nietrywialne i nie istnieje jedna przenośna i skuteczna metoda ochrony. Jednym z rozwiązań jest wykonywanie operacji sprawdzenia uprawnień i ich użycia w sposób atomowy, aby nie mogły być przerwane przez inne operacje.

Problem niebezpiecznych wyścigów daje o sobie znać na różne sposoby. Przykładowo w Internet Explorerze 6 istniał błąd w zarządzaniu pamięcią, który powodował, że jeden wątek mógł odczytywać pamięć, która została nadpisana przez inny wątek lub przez niego zwolniona (CAN-2005-0553). Atakujący mógł to wykorzystać do zdalnego wykonania dowolnego kodu.

Null pointer dereference

Ponieważ przełączanie kontekstu w wielozadaniowym systemie operacyjnym jest operacją kosztowną, nie stosuje się przełączania przestrzeni adresowej przy przechodzeniu pomiędzy kodem przestrzeni użytkownika a przestrzeni jądra. Z tego powodu przestrzeń adresowa jądra jest zmapowana w przestrzeni adresowej każdego procesu użytkownika. Mechanizm ochrony pamięci zapobiega nadpisaniu pamięci jądra przez aplikacje. Problem jednak pojawia się w przypadku tzw. pustych wskaźników (NULL pointer). Jeśli aplikacja zaalokuje stronę pamięci pod adresem 0x00000000 to ten adres stanie się adresem prawidłowym i próba dostępu do pamięci poprzez pusty wskaźnik nie będzie generować błędu. Aplikacja będzie mogła korzystać z adresu zerowego tak samo jak z innych adresów należących do zaalokowanych obszarów.

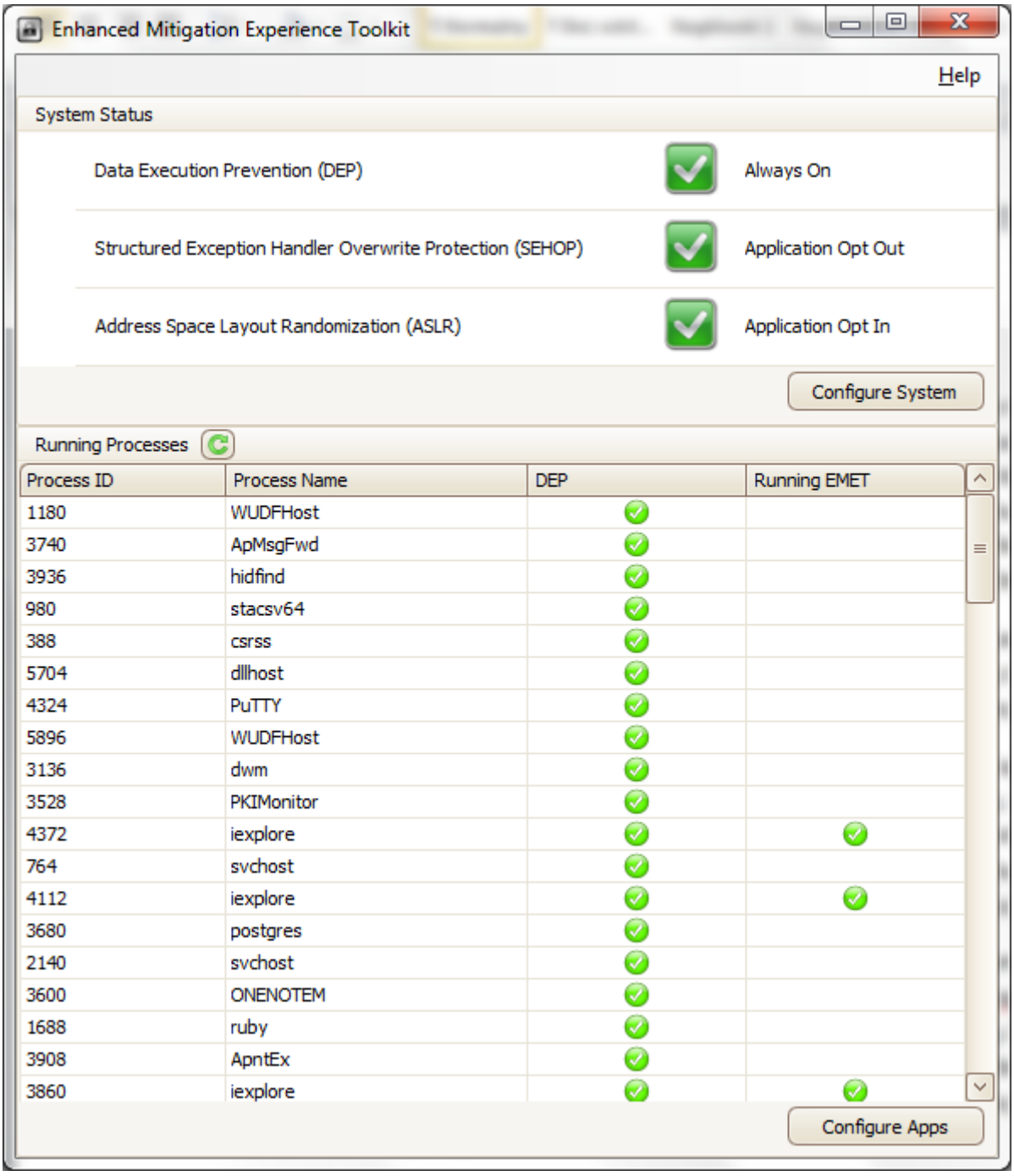
Niebezpieczeństwo bierze się z faktu, że w jądrze może zdarzyć się błąd powodujący użycie pustego wskaźnika. Np. w jądrze Linuksa bardzo często używa się wskaźników do funkcji i o taki błąd nie jest trudno. Jeśli pamięć pod adresem 0x00000000 zostanie zaalokowana i nadpisana danymi przez atakującego z poziomu aplikacji, jądro odwoła się do tych danych przez pusty wskaźnik. Atakujący może w ten sposób przekazać do jądra np. wskaźnik do stworzonej przez siebie funkcji i spowodować jej wykonanie z uprawnieniami jądra systemu operacyjnego, uzyskując tym samym nieograniczone uprawnienia. Takie dziury nie są obce też systemowi Windows, np. MS09-058, MS09-065, MS11-054 czy MS11-077.

Zabezpieczeniem wprowadzonym w Windows 8 w celu rozwiązania tego problemu jest uniemożliwienie procesom z przestrzeni użytkownika alokowania dolnych 64 kB pamięci.

Enhanced Mitigation Experience Toolkit

EMET to narzędzie firmy Microsoft zapewniające dodatkową ochronę aplikacji bez potrzeby ich rekompilacji. Pozwala dla poszczególnych procesów włączyć takie mechanizmy jak:

- SEHOP
- DEP
- Zabezpieczenie przed heap spray
- Alokacja zerowej strony pamięci
- Mandatory ASLR
- EAF
- Bottom-up randomization



Aby wprowadzać zabezpieczenia w aplikacjach EMET korzysta z tzw. shims. Jest to metoda wykorzystywana przez funkcję zgodności aplikacji w Windows, dzięki której można wpływać na programy tak, aby działały poprawnie pod nowymi wersjami Windows bez potrzeby rekompilacji. Dzięki shims można przechwytywać odwołania do funkcji API i w ten sposób oszukać program m.in. co do wersji Windows, posiadanych uprawnień czy też przekierować dostęp do plików. W przypadku EMET ładowana jest biblioteka EMET.dll do każdego procesu chronionego przez EMET.

Aby włączyć DEP dla chronionego procesu, załadowana biblioteka EMET.dll uruchamia wewnętrzną funkcję aSetprocessdepp(), która z kolei wywołuje funkcję SetProcessDEPPolicy() z biblioteki kernel32.dll. W ten sposób DEP jest włączony dynamicznie, podczas uruchamiania procesu.

Zabezpieczenie przed heap spray polega na alokacji pamięci pod adresami, które są zwykle wykorzystywane w tej technice, domyślnie są to adresy: 0x0a040a04, 0x0a0a0a0a, 0x0b0b0b0b, 0x0c0c0c0c, 0x0d0d0d0d, 0x0e0e0e0e, 0x04040404, 0x05050505, 0x06060606, 0x07070707, 0x08080808, 0x09090909, 0x14141414. Ogranicza to liczbę adresów, w które atakujący może próbować się „wstrzelić" aby trafić na swój shellcode umieszczony w pamięci przez heap spray.

W podobny sposób alokacja zerowej strony pamięci ma zabezpieczyć przed atakami z wykorzystaniem odwołań za pomocą pustych wskaźników (Null pointer dereference). Jest to jednak funkcja, która tak naprawdę nie pełni obecnie żadnej roli i została dodana z myślą, że może kiedyś okazać się przydatna. Należy przypomnieć, że problem Null pointer dereference dotyczy jądra a EMET chroni aplikacje przestrzeni użytkownika, nie jądro. Dla tych aplikacji nie ma jeszcze w tej chwili metody wykorzystania pustych wskaźników do ich atakowania.

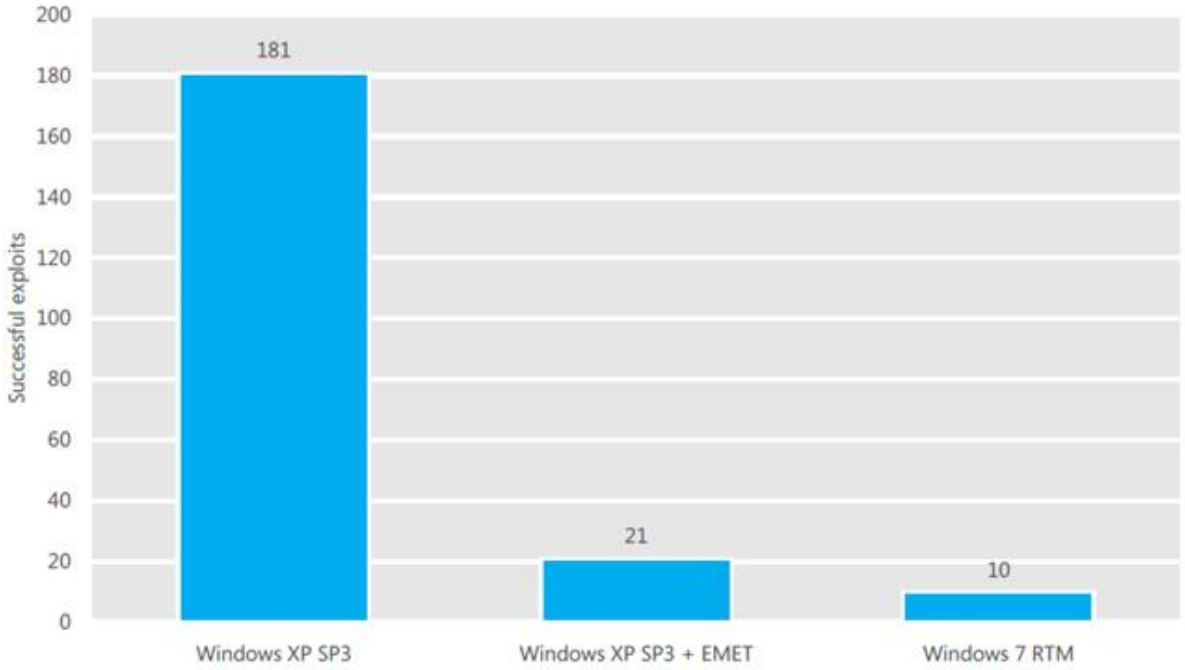
Mandatory ASLR jest funkcją przeznaczoną dla aplikacji, które nie są kompatybilne z ASLR i w związku z tym nie opiera się na systemowej funkcji ASLR. Zamiast tego alokuje losową liczbę (od 0 do 256) 64-kilobajtowych bloków pod adresem, który jest adresem bazowym danego modułu. Przez to zmusza Windows do przesunięcia adresu bazowego tego modułu pod pierwszy wolny adres. W ten sposób uzyskiwane są losowe adresy bazowe, podobnie jak w normalnym ASLR. W praktyce jednak okazuje się, że uzyskiwana entropia jest dosyć niska i wynosi ok. 4 bitów.

Export Address Table Access Filtering (EAF) sprawdza, czy odwołanie to tabeli eksportów funkcji nastąpiło z kodu należącego do załadowanych modułów. Ma w ten sposób zapobiegać dostępowi do EAT shellcode'owi znajdującemu się np. na stosie lub stercie, który chciałby przeszukać tabelę eksportów po to aby wykonywać funkcje WinAPI. EAF może w ten sposób zabezpieczyć przed wieloma exploitami wywołującymi systemowe funkcje API, jest jednak dosyć prosty do obejścia. Przykładowo można wykorzystać ROP żeby odczytać EAT za pomocą kodu pochodzącego z jednego z załadowanych modułów, a więc uznawanego przez EAF za zaufany.

Funkcja bottom-up randomization wprowadza losowość adresów stosu, sterty oraz innych alokowanych obszarów. Co ciekawe, zwiększa też entropię funkcji Mandatory ASLR.

Nie wszystkie aplikacje są kompatybilne z zabezpieczeniami wprowadzanymi przez EMET. Z tego powodu niektóre z zabezpieczeń muszą być czasem wyłączone dla niektórych procesów. Aby skrócić czas wdrożenia EMET i ograniczyć liczbę koniecznych testów, Microsoft dostarcza pliki XML zawierające gotową konfigurację dla najpopularniejszych swoich aplikacji a także produktów firm zewnętrznych.

Skuteczność EMETa okazuje się całkiem spora. Poniżej testy 184 exploitów wykorzystujących podatności w aplikacjach.



EMET dostępny jest obecnie w wersji 3.0. Testowana jest wersja 3.5, w której dodano ochronę przed exploitami korzystającymi z ROP:

- Wyłączone jest ładowanie bibliotek DLL z udziałów SMB
- Wyłączone jest oznaczanie obszaru stosu jako wykonywalnego
- Przy wywoływaniu krytycznych, często używanych w exploitach funkcji, sprawdzane jest, czy wywołanie nastąpiło instrukcją CALL czy RET. W tym drugim przypadku następuje zatrzymanie działania aplikacji jako będącej celem ataku.
- Przy wywoływaniu krytycznych funkcji sprawdzane jest, czy nie są potem wywoływane tzw. ROP gadgets. W tym celu przeprowadzana jest emulacja instrukcji znajdujących się pod adresem powrotnym z wołanej funkcji.
- Wykrywanie tzw. stack pivoting, czyli przestawienia wskaźnika stosu (rejestr ESP) aby pokazywał na shellcode.

Oczywiście także te zabezpieczenia nie dają 100% ochrony i też można je obejść, np. wywołując funkcje systemowe za pomocą KiFastSystemCall() i włączając wykonywalność stosu za pomocą ZwProtectVirtualMemory().