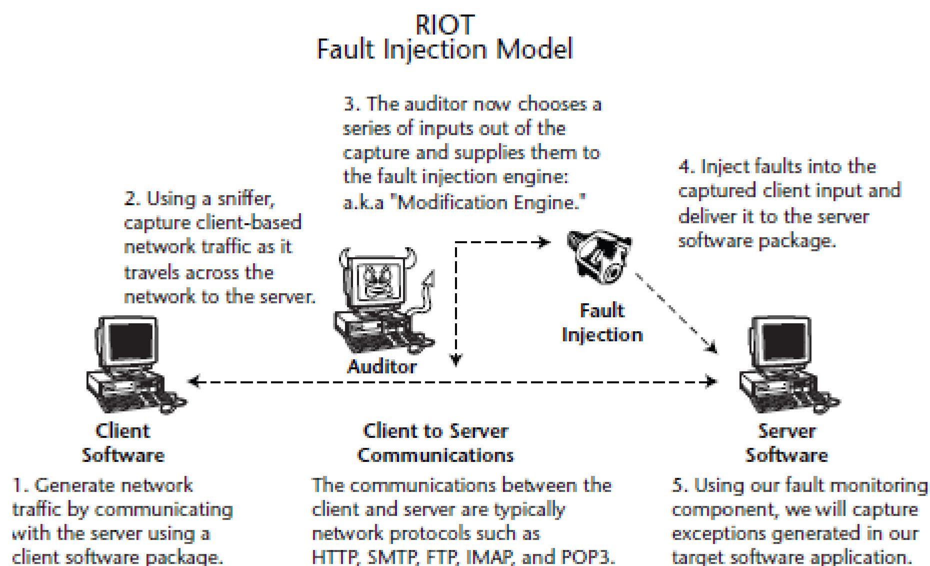
Wstrzykiwanie błędów

Technologie wstrzykiwania błędów są używane od ponad pół wieku do weryfikacji odporności rozwiązań sprzętowych na awarie. Uszkodzone systemy wtryskowe są obecnie używane do testowania maszyn w samochodach, którymi jeździmy, silników w samolotach, które nas latają, a nawet elementów grzewczych, które podgrzewają naszą kawę. Systemy te wprowadzają błędy przez styki obwodów scalonych, poprzez impulsy EMI, zmieniając poziomy napięcia, w niektórych przypadkach nawet przez zastosowanie promieniowania. W dzisiejszych czasach każdy liczący się producent sprzętu stosuje w swoim procesie testowania jakiś system wstrzykiwania błędów. Ponieważ nasze technologie przechodzą z technologii analogowej na cyfrową, ilość używanego oprogramowania rośnie w tempie wykładniczym. Pytanie, które należy zadać, brzmi: jakie mamy narzędzia, które przetestują niezawodność naszego oprogramowania? W ciągu ostatniej dekady opracowano kilka rozwiązań do wstrzykiwania błędów w celu wykrywania poważnych problemów w oprogramowaniu dla przedsiębiorstw. Wiele z tych rozwiązań do wstrzykiwania błędów opartych na oprogramowaniu zostało stworzonych w trakcie kilku grantów badawczych sponsorowanych przez Office of Naval Research (ONR), Defense Advanced Research Project Agency (DARPA), National Science Foundation (NSF) i Digital Equipment Corporation (GRUDZIEŃ). Systemy wstrzykiwania błędów oprogramowania, takie jak DEPEND, DOCTOR, Xception, FERRARI, FINE, FIST, ORCHESTRA, MENDOSUS i ProFI wykazały, że technologie wstrzykiwania błędów mogą być wykorzystywane do skutecznego wyliczania różnych błędów w aplikacjach oprogramowania dla przedsiębiorstw Każde z kilku z tych rozwiązań zaprojektowany, aby pomóc rozwiązać ten sam problem - zaoferować społeczności programistów zasoby, które pozwolą im przetestować odporność na awarie swojego oprogramowania. Niewiele rozwiązań w sektorze publicznym i prywatnym zostało zaprojektowanych specjalnie do wykrywania luk w zabezpieczeniach w docelowym oprogramowaniu. Ponieważ znaczenie bezpieczeństwa rośnie z dnia na dzień, rośnie również zapotrzebowanie na technologie, które pomogą poprawić bezpieczeństwo używanego przez nas oprogramowania. Narzędzia do testowania usterek są codziennie używane przez inżynierów Działu Jakości (QA) do testowania przypisanego im oprogramowania pod kątem potencjalnych słabości. Jedną z najbardziej przydatnych umiejętności, jakie mogą posiadać inżynierowie QA, jest umiejętność włączenia automatyzacji do swoich zestawów narzędzi. Audytorzy bezpieczeństwa oprogramowania mogliby się wiele nauczyć z nowoczesnych technik zapewniania jakości. Większość utalentowanych audytorów bezpieczeństwa polega na ręcznych technikach audytu, głównie inżynierii wstecznej i audytach źródeł, aby wykryć potencjalne problemy z zabezpieczeniami w produktach programowych. Chociaż umiejętności te są przydatne, jeśli nie są wymagane, w przypadku udanego audytora, ważna jest również umiejętność rozwijania technologii automatycznego audytu. Korzystając z wiedzy odkrytej podczas wycofywania, testerzy oprogramowania mogą szybko skonfigurować swoje aplikacje audytowe do audytowania oprogramowania podczas wykonywania innych zadań audytowych. Ten rodzaj wielozadaniowości pozwala audytorowi na wykonanie pracy setek, jeśli nie tysięcy innych audytorów oprogramowania w ułamku czasu. Jednym z najlepszych aspektów testowania błędów jest to, że każdy błąd, który popełnisz podczas tworzenia rozwiązania, może w rzeczywistości zwiększyć sukces testowania. Błąd w twoim rozwoju to jedna z najbardziej nieoczekiwanych rzeczy, które możesz zrobić. Jeśli cofniesz się i sporządzisz listę wszystkich błędów programistycznych, które popełniłeś w czasie, i wbudujesz test dla każdego z nich w swojej aplikacji do testowania błędów, możesz łatwo zepsuć większość produktów oprogramowania serwerowego dla przedsiębiorstw. Zbudowanie rozwiązania do wstrzykiwania błędów zmotywuje Cię do nauczenia się klas ataków do takiej głębokości, że zrozumiesz je na znacznie prostszym poziomie. Z każdą nową klasą ataku, którą poznasz lub odkryjesz, nauczysz się sztuczek i technik, które pomogą ci zrozumieć inne klasy. To, czego się nauczysz, może jeszcze bardziej zwiększyć możliwości Twojego pakietu audytowego. Najlepsze jest to, że korzystając z automatyzacji, możesz nawet podczas snu znaleźć

wstrząsające światem luki w zabezpieczeniach. W tej części zaprojektujemy i wdrożymy rozwiązanie polegające na wstrzykiwaniu błędów w celu wykrycia luk w zabezpieczeniach w produktach oprogramowania serwera sieciowego, które działają na nośniku sieciowym opartym na protokole aplikacji. Ten system wstrzykiwania błędów, który nazwiemy RIOT, bardzo przypomina system zaprojektowany w styczniu 2000 roku, który został wykorzystany do wykrycia kilku szeroko nagłośnionych luk w zabezpieczeniach, takich jak te wykorzystywane przez wirusa Code Red. Korzystając z RIOT, demonstrujemy skuteczność testowania błędów, wyliczając niektóre z tych luk bezpieczeństwa w naszej docelowej aplikacji, Internet Information Server (IIS) 5.0 firmy Microsoft.

Przegląd projektu

Elementy składowe naszego systemu wstrzykiwania usterek pokazano na rysunku 16-1.



Większość systemów wstrzykiwania usterek można podzielić i skategoryzować w podobny sposób. Omówimy szczegółowo każdy z tych elementów w tym rozdziale; później połączymy elementy i zbudujemy RIOT.

Generowanie danych wejściowych

Do zbierania danych wejściowych do wstrzykiwania błędów można użyć różnych mediów. W tej sekcji wymienimy kilka, ale jak się przekonasz, dostępnych jest wiele innych. Nasz wkład można podzielić na różne suplementy testowe. Każdy suplement zapoczątkuje generację serii testów. Ilość i rodzaj danych w naszym wejściu określi, jakie testy zostaną wykonane. Chociaż nasz wkład można zebrać bez względu na jego zawartość, nasza skuteczność w odkrywaniu błędów w naszym docelowym oprogramowaniu drastycznie wzrosną, jeśli dostarczymy dane wejściowe, które zostały użyte do komunikacji z ezoterycznymi i nieprzetestowanymi funkcjami oprogramowania. W naszych przykładach skupimy się na danych wejściowych protokołu aplikacji, takich jak stan pierwszego klienta transakcji HTTP. Moglibyśmy zacząć zbierać dane wejściowe do naszych testów, przechwytując ruch sieciowy z sesji przeglądarki na produkcyjnym serwerze sieci Web. Załóżmy, że podczas monitorowania przechwyciliśmy następujące żądanie klienta ruch w sieci lokalnej:

GET /search.ida?group=kuroto&q=riot HTTP/1.1

Zaakceptować: */*

Akceptuj-język: en-us

Akceptuj-kodowanie: gzip, deflate

Klient użytkownika: Mozilla/4.0

Host: 192.168.1.1

Połączenie: Keep-Alive

Plik cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG

Ktoś zaznajomiony z protokołem HTTP na poziomie średniozaawansowanym może zauważyć, że rozszerzenie .ida nie jest standardowym rozszerzeniem pliku. Po przeprowadzeniu niewielkich badań za pomocą naszej ulubionej wyszukiwarki odkrywamy, że to rozszerzenie jest częścią słabo udokumentowanej funkcji dostępnej za pośrednictwem filtra ISAPI zainstalowanego z wieloma wersjami serwera internetowego IIS.

UWAGA: Każda funkcja, która jest trudna do nauczenia, trudna w użyciu i trudna do lubienia, jest doskonałym miejscem do rozpoczęcia poszukiwania problemów związanych z bezpieczeństwem. Jeśli ta funkcja odwraca twoją uwagę od podstawowej funkcjonalności programu, najprawdopodobniej miała taki sam wpływ na programistów i testerów — zanim została wyrzucona, by sprostać wymaganiom wytrwałych klientów.

Powyższy przykład zostanie dostarczony do komponentu wstrzykiwania błędów naszej aplikacji testowej. Ten komponent wstrzykujący błędy wprowadzi błędy (złe lub nieoczekiwane dane wejściowe) do danych wejściowych poprzez ich modyfikację. Dane wejściowe, które dostarczamy do naszego komponentu do wstrzykiwania usterek, w znacznym stopniu wpłyną na spektrum naszych testów. Jakość naszego wkładu będzie również miała duży wpływ na nasze testy. Jeśli dostarczymy dane wejściowe, które są bezpośrednio nieprawidłowe, spędzimy większość czasu na kontrolowaniu procedur obsługi błędów w naszej aplikacji docelowej. Z tego powodu chcemy poświęcić znaczną ilość czasu na staranne zbieranie naszych danych wejściowych. W zależności od ilości zebranych danych wejściowych, możemy również chcieć ręcznie zweryfikować ich jakość, zanim rozpoczniemy test na pełną skalę. Do zbierania danych wejściowych, które możemy dostarczyć do rozwiązania wstrzykiwania usterki, można użyć różnych metod. Wybrana przez nas metoda wprowadzania lub kombinacja metod będzie zależeć od rodzaju przeprowadzanego przez nas testowania usterek.

Generowanie ręczne

Ręczne generowanie danych wejściowych może być bardzo czasochłonne, ale zazwyczaj daje najlepsze wyniki. Możemy ręcznie utworzyć nasze dane wejściowe za pomocą wybranego przez nas edytora, zapisując każdy utworzony test jako osobny plik w katalogu. Możemy napisać prostą funkcję w naszym programie, aby zbadać ten katalog i odczytać każde z wejść testowych, przekazując je pojedynczo do naszego komponentu do wstrzykiwania błędów. Użyjemy tej metody w naszej przykładowej aplikacji do wstrzykiwania błędów RIOT. Moglibyśmy również przechowywać stworzone przez nas dane wejściowe w bazie danych lub uwzględnić je bezpośrednio w naszej aplikacji. Zapisywanie danych wejściowych bezpośrednio do pliku oszczędza nam kłopotów z tworzeniem niestandardowych struktur danych w celu ich organizowania, rejestrowania ich rozmiaru i obsługi ich zawartości.

Automatyczne generowanie

W przypadku prostych protokołów, takich jak HTTP, możemy chcieć wygenerować nasze dane wejściowe. Możemy to zrobić, studiując protokół i projektując algorytm do generowania potencjalnych

danych wejściowych. Generowanie danych wejściowych jest niezwykle przydatne w przypadkach, w których chcemy przetestować duży zakres protokołu, ale nie chcemy ręcznie tworzyć wszystkich danych wejściowych. Podczas moich doświadczeń z testowaniem odkryłem, że automatyczne generowanie danych wejściowych było pomocne w przypadku prostych protokołów, które miały bardzo niezawodną strukturę, takich jak większość protokołów aplikacji. Podczas pracy z protokołami o znacznie bardziej dynamicznym charakterze, oferującymi wiele warstw i kilka stanów, automatyczne generowanie danych wejściowych może nie być optymalną metodą wprowadzania danych. Błąd w automatycznym generowaniu danych wejściowych może pojawić się dopiero kilka godzin po rozpoczęciu testowania. Jeśli nie monitorujesz ściśle danych wejściowych podczas ich generowania, możesz nie zauważyć, że wygenerowane dane wejściowe są problematyczne.

Przechwytywanie na żywo

Kilka rozwiązań, takich jak ORCHESTRA, oferuje możliwość wstrzykiwania błędów bezpośrednio do istniejących protokołów komunikacyjnych. Ta metoda jest bardzo skuteczna podczas testowania złożonych protokołów opartych na stanach. Jedyną wadą jest wymóg zdefiniowania przez użytkownika protokołu, tak aby można było wprowadzić zmiany gwarantujące pomyślne dostarczenie danych. Na przykład, jeśli zmienisz rozmiar danych w komunikacie protokołu, może być konieczne zaktualizowanie pól o różnych długościach w celu odzwierciedlenia dokonanych zmian. Jedną z nielicznych grup, które przezwyciężyły ten podobny problem, była grupa badaczy, którzy opracowali ORCHESTRA; użyli odgałęzień protokołu, aby zdefiniować niezbędne cechy protokołu.

Generowanie „Fuzz"

Pod koniec lat 80. i na początku lat 90. trzech badaczy - Barton Miller, Lars Fredriksen i Bryan So - przeprowadziło badanie integralności popularnych narzędzi wiersza poleceń systemu Unix. Pewnej nocy podczas burzy z piorunami jeden z badaczy próbował użyć niektórych standardowych narzędzi uniksowych przez połączenie dial-up. Z powodu szumu linii pozornie losowe dane były wysyłane do narzędzi uniksowych zamiast tego, co pisał w swojej powłoce. Zauważył, że wiele programów robiło zrzut rdzenia, gdy próbował ich użyć z powodu tych losowych danych. Korzystając z tego odkrycia, trzej badacze opracowali fuzz, program przeznaczony do generowania pseudolosowych danych wejściowych, które można wykorzystać do testowania integralności ich aplikacji. Generowanie Fuzz-input stało się teraz częścią wielu pakietów do wstrzykiwania błędów. Jeśli chcesz dowiedzieć się więcej o fuzz, odwiedź archiwum na http://www.cs.wisc.edu/~bart/fuzz/. Wielu obecnych audytorów uważa, że używanie danych wejściowych fuzz jest jak strzelanie do nietoperzy w ciemności. W trakcie projektu fuzz ci trzej badacze odkryli przepełnienia liczb całkowitych, przepełnienia buforów, błędy formatowania i ogólne problemy z analizatorem składni w wielu różnych zastosowaniach. Należy zauważyć, że kilka z tych klas ataków zostało publicznie znanych i zaakceptowanych dopiero ponad dekadę po tych badaniach.

Błąd wtrysku

W poprzedniej sekcji omówiliśmy metody generowania danych wejściowych, które będą używane przez nasz komponent do wstrzykiwania błędów. W tej sekcji omówimy modyfikacje, które możemy wprowadzić na naszych danych wejściowych, które będą generować błędy, takie jak wyjątki, w aplikacji, którą będziemy testować. Ta faza procesu jest tym, co naprawdę definiuje rozwiązanie. Chociaż metody używane do zbierania danych wejściowych pozostają podobne we wszystkich rozwiązaniach wstrzykiwania błędów, metody używane do wstrzykiwania błędów i rodzaje wstrzykiwanych błędów są radykalnie różne. Niektóre rozwiązania wstrzykiwania błędów wymagają dostępu do źródła, aby można było wprowadzać modyfikacje w testowanym programie, które umożliwią audytorowi zbieranie informacji w czasie wykonywania. Ponieważ nasz pakiet do

wstrzykiwania błędów jest ukierunkowany na aplikacje o zamkniętym kodzie źródłowym, nie będziemy musieli w żaden sposób modyfikować aplikacji; zmodyfikujemy tylko dane wejściowe, które normalnie są przekazywane do aplikacji docelowej.

Silniki modyfikacji

Po przetworzeniu zebranych danych wejściowych i przesłaniu ich do naszego silnika modyfikacji, możemy rozpocząć wstawianie błędów do danych wejściowych. Będziemy musieli przechowywać w pamięci pierwotną kopię danych wejściowych, którą będziemy mogli pozyskiwać, modyfikować i dostarczać dla każdej iteracji naszego silnika. W tym przypadku iteracja jest po prostu sekwencją wstrzykiwania błędu i dostarczenia zmodyfikowanych danych wejściowych do aplikacji docelowej. Silnik modyfikacji próbek jest nastawiony na wykrywanie luk w zabezpieczeniach przepełnienia bufora. Ten silnik podzieli strumień wejściowy na elementy, wstawi błąd do każdego elementu (w tym przypadku bufor danych o zmiennej wielkości) i na koniec wyśle go do aplikacji docelowej. Przykładowy silnik, którego użyjemy, różni się od innych opracowanych wcześniej systemów wtrysku usterek. Zamiast na ślepo wstawiać błędy w sposób sekwencyjny, nasz silnik przeanalizuje dane wejściowe i określi, gdzie wstawić błędy na podstawie zawartości danych wejściowych. Przykładowy silnik będzie również naśladował wstawione błędy do otaczających go danych, dzięki czemu podczas naszych audytów nie będziemy omijać nas przez typowe schematy oczyszczania danych wejściowych. Te i inne różnice znacznie zwiększą wydajność naszych testów usterek. Jeśli kiedykolwiek pisałeś aplikację do wstrzykiwania błędów, prawdopodobnie przeszukiwałeś dane sekwencyjnie, wstrzykiwałeś błędy i dostarczałeś je do docelowej aplikacji. Bez jakiejkolwiek optymalizacji logiki wstrzykiwania błędów, prawdopodobnie zauważyłeś, że każda sesja testowa wymagała dużo czasu na ukończenie i że przeprowadzono wiele niepotrzebnych testów. Dokonując kilku prostych zmian w naszej logice wstrzykiwania błędów, możemy radykalnie zminimalizować liczbę testów, które należy wykonać. Weźmy przykładowy strumień wejściowy:

GET /index.html HTTP/1.1

Host: test.com

Założymy, że dopiero rozpoczęliśmy test, a nasz indeks znajduje się na pierwszym bajcie naszej metody HTTP G. Podczas pierwszej iteracji wstawimy błąd na tej pozycji. Następnie dostarczymy te zmodyfikowane dane wejściowe do naszego celu. Po zakończeniu dostawy wstawimy nasz następny błąd w tym samym miejscu i dostarczymy zmodyfikowane dane wejściowe. Będzie to trwało, dopóki nie przejdziemy przez możliwe błędy, które możemy wstrzyknąć. Następnym krokiem jest przeniesienie naszego indeksu na następną pozycję, drugi bajt metody HTTP lub E. Powtórzymy modyfikację i dostarczymy każdy błąd, tak jak zrobiliśmy to z poprzednią pozycją indeksu. Innymi słowy, dla każdego stanowiska dokonamy modyfikacji dla każdej oferowanej przez nas usterki. Jeśli mamy 10 wejść, każde z 5000 możliwych pozycji, a nasz silnik ma 1000 usterek, to po wykonaniu tego testu powinniśmy być w stanie kupić latający samochód. Zamiast sekwencyjnie wstawiać błędy w całym strumieniu wejściowym, możemy podzielić nasze dane na elementy za pomocą logiki ograniczników. Następnie wstawiamy nasze błędy w przesunięciu każdego elementu zamiast w każdym przesunięciu w danych wejściowych. Poprzedni przykładowy strumień wejściowy ma metodę, identyfikator URI, wersję protokołu, nazwę nagłówka i wartość nagłówka. Aby jeszcze bardziej to zepsuć, powinniśmy również zwrócić uwagę na rozszerzenie pliku w adresie URL, główne i podrzędne wersje protokołu, a nawet kod kraju lub główny DNS nazwy hosta. Ręczne budowanie wsparcia w naszym pakiecie audytowym dla każdego elementu każdego protokołu, który chcemy przetestować, jest strasznym zadaniem. Na szczęście istnieje o wiele prostszy sposób, aby to osiągnąć.

Rozgraniczanie logiki

Kiedy programiści tworzą swoje parsery, rzadko, jeśli w ogóle, tworzą znaczniki, które pasują do systemu alfabetycznego lub numerycznego. Znaczniki to zwykle widoczne symbole, takie jak # lub $. Aby wyjaśnić tę koncepcję, spójrzmy na poniższe. Jeśli użyliśmy wartości 1 do oddzielenia wersji głównej i pomocniczej protokołu, jak moglibyśmy określić wersję protokołu w następującym strumieniu wejściowym?

GET /index.html HTTP/111

Gdybyśmy użyli wartości alfanumerycznych jako symboli analizy, jak moglibyśmy nazwać lub opisać nasze dane? Informacje możemy odczytać za pomocą specjalnych symboli, takich jak kropka w tym przypadku:

GET /index.html HTTP/1.1

Podstawą komunikacji jest równowaga i częstotliwość dystrybucji i separacji elementów w informacji. Wyobraź sobie, jak trudno byłoby przeczytać ten rozdział, gdybyśmy usunęli wszystkie wartości niealfanumeryczne – bez spacji, bez kropek, tylko litery i cyfry. Jedną ze wspaniałych rzeczy w ludzkim umyśle jest to, że możemy podejmować decyzje na podstawie tego, czego nauczyliśmy się przez całe życie. Moglibyśmy więc przyjrzeć się temu niezorganizowanemu bałaganowi informacji i z czasem określić, co jest ważne, a co nie. Niestety, oprogramowanie używane przez naszą infrastrukturę nie jest tak inteligentne. Musimy sformatować nasze informacje przy użyciu odpowiedniego standardu protokołu, abyśmy mogli komunikować się z odpowiednim oprogramowaniem. Sformatowane dane używane w protokołach aplikacji opierają się przede wszystkim na koncepcji delimitacji. Ograniczniki są zwykle drukowanymi wartościami ASCII, które nie są alfanumeryczne. Przyjrzyjmy się jeszcze raz przykładowemu strumieniowi wejściowemu; tym razem uciekniemy od znaków normalnie niewidocznych przez \:

GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

Zauważ, że każdy element w strumieniu wejściowym protokołu próbki jest oddzielony ogranicznikiem. Metoda jest oddzielona spacją, identyfikator URI jest oddzielony spacją, nazwa protokołu jest oddzielona ukośnikiem, główna wersja kropką, podrzędna znakiem powrotu karetki i nowym wierszem; nazwa nagłówka jest oddzielona dwukropkiem, a po niej następuje wartość nagłówka, oddzielona dwoma znakami powrotu karetki i nowymi wierszami. Tak więc, po prostu umieszczając nasze błędy wokół specjalnych symboli w naszym strumieniu wejściowym, możemy przetestować błędy prawie każdego elementu protokołu, nie wiedząc nic o nich. Powinniśmy wstawić nasze błędy przed i po tych specjalnych symbolach, aby nie mieć problemów z audytem przypisań lub granic w naszym strumieniu wejściowym. Próbny przebieg z dziesięcioma iteracjami przy użyciu błędu EEYE2003 wytworzyłby następujące uszkodzone strumienie wejściowe.

Sekwencyjny wtrysk błędu:

EEYE2003GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GEEYE2003ET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GEEEYE2003T /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /iEEYE2003ndex.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /inEEYE2003dex.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /indEEYE2003ex.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /indeEEYE2003x.html HTTP/1.1\r\nHost: test.com\r\n\r\n

Wstrzyknięcie błędu za pomocą logiki ogranicznika:

GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /indexEEYE2003.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.EEYE2003html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.htmlEEYE2003 HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html EEYE2003HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HTTPEEYE2003/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HTTP/EEYE20031.1\r\nHost: test.com\r\n\r\n

Widzimy wzrost wydajności nawet w tym małym strumieniu wejściowym. W systemie, który wykorzystuje średnio kilka tysięcy strumieni danych, z niemal nieskończoną liczbą możliwych błędów, ta optymalizacja może zaoszczędzić nam tygodnie, miesiące, jeśli nie lata testowania. Każdy może napisać coś, co w ciągu kilku lat znajdzie lukę w zabezpieczeniach; bardzo niewielu potrafi napisać coś, co zrobi to w pięć minut.

Poruszanie się po oczyszczaniu danych wejściowych

Teraz, gdy omówiliśmy, gdzie będziemy wstawiać nasze błędy, porozmawiajmy o rzeczywistych błędach, które będziemy wstawiać. Załóżmy, że w naszym pierwszym silniku modyfikacji skoncentrowanym na wykrywaniu luk w zabezpieczeniach przepełnienia bufora wybraliśmy jeden błąd. Naszym jedynym błędem jest 1024-bajtowy bufor wypełniony znakiem X. Używając tego jednego błędu, możemy znaleźć kilka problemów w naszym docelowym pakiecie oprogramowania, ale jest to bardzo mało prawdopodobne z powodu ograniczenia rozmiaru i zawartości. Jeśli nie uda nam się uzyskać danych z wprowadzonymi błędami po początkowym oczyszczeniu danych wejściowych, które wykonuje nasze oprogramowanie docelowe, większość czasu testowego spędzimy na odbijaniu się od procedur obsługi błędów. Nasze aplikacje docelowe zazwyczaj ograniczają początkowy rozmiar każdego elementu protokołu. Na przykład metoda HTTP może być ograniczona do 128 bajtów, ale później metoda pobiera strumień wejściowy i kopiuje go do statycznego bufora 32 bajtów. Ponieważ błąd, który wybraliśmy do wstrzyknięcia to 1024 bajty (znacznie powyżej 128 bajtów), docelowa aplikacja porzuci strumień wejściowy i zwróci błąd. Nasza wina nigdy nie zostanie dostarczona do wrażliwego bufora. Moglibyśmy użyć spektrum rozmiarów buforów, na przykład od 1 do 1024, z przyrostami co 1 bajt. Biorąc pod uwagę strumień wejściowy składający się z kilkuset elementów, z których każdy wstrzykniemy 1024 możliwych błędów, czas potrzebny na wykonanie tego typu testu może być nierozsądny. Dlatego spektrum automatycznie generowanych rozmiarów nie jest najwłaściwsze. Mając do czynienia z aplikacjami o zamkniętym kodzie źródłowym, często można się

wiele nauczyć, obserwując, w jaki sposób programiści zaimplementowali określone struktury danych, takie jak rozmiary buforów. Podczas inspekcji serwera HTTP o zamkniętym kodzie źródłowym można przeanalizować kod źródłowy kilku pakietów serwerów typu open source, takich jak Apache, Sendmail i Samba. Przeszukując źródło, możesz określić, jakie typowe rozmiary buforów są używane. Odkryjesz, że większość rozmiarów buforów to wielokrotności potęg 2, zaczynając od 32; na przykład 32, 64, 128, 256, 512, 1024 i tak dalej. Inne to potęgi liczby 10. Pozostałe są oparte na tym samym schemacie, ale mają zmienną liczbę dodawaną lub odejmowaną od nich. Ta liczba o zmiennej wielkości wynosi zwykle od 1 do 20. Korzystając z tych statystyk, można utworzyć tabelę rozmiarów buforów, które potencjalnie mogą wywołać większość luk w zabezpieczeniach związanych z przepełnieniem bufora. Dodaj małą deltę przed i po rozmiarach buforów, aby uwzględnić wszelkie typowe dodatki zauważone w deklaracjach zmiennych. Skuteczną metodą potwierdzenia, że mamy dobre dane wejściowe o błędach, jest przeprowadzenie testów na podatnym oprogramowaniu, o którym wiadomo, że zawiera określone luki w zabezpieczeniach związane z przepełnieniem bufora. Korzystając z tabeli rozmiarów buforów, przekonasz się, że tabela danych wejściowych będzie w stanie odtworzyć każde przepełnienie bufora w oprogramowaniu docelowym. Zamiast 70 000 możliwych wstrzyknięć błędów na element protokołu, mamy teraz około 800. Aplikacje oprogramowania dla przedsiębiorstw często weryfikują zawartość wejściową przed przejściem do wewnętrznych procedur, aby uniknąć potencjalnych problemów. To zachowanie nie wynika bezpośrednio z bezpiecznego programowania, ale sprawia, że wykrywanie i wykorzystywanie luk w zabezpieczeniach jest nieco trudniejsze. Jeśli chcemy przeprowadzić audyt ciemniejszych zakamarków oprogramowania, w których istnieje wiele nieodkrytych luk w zabezpieczeniach, będziemy musieli obejść różne ograniczenia dotyczące treści. Często pola będą ograniczone do cyfr, wielkich liter lub schematów kodowania. Funkcje C isdigit(), isalpha(), isupper(), islower() i isascii() są zwykle używane do ograniczania zawartości. Jeśli wprowadzimy błąd, który zawiera dane nieliczbowe do elementu protokołu, który może przechowywać tylko dane liczbowe, oprogramowanie zwróci błąd po zauważeniu, że wywołanie isdigit() nie powiodło się. Wprowadzając błędy, które odzwierciedlają otaczające dane, możemy obejść większość tych ograniczeń. Możemy wstrzyknąć błąd wypełniony wartością bajtu otaczającego elementu protokołu. Porównajmy normalną sesję wstrzykiwania błędów z sesją wstrzykiwania błędów, która wstrzykuje błędy w celu odzwierciedlenia otaczających danych. Próbne uruchomienie z buforem o rozmiarze 10 dałoby następujące strumienie wejściowe wstrzykiwane błędy:

GETTTTTTTTT /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET ///////////index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /iiiiiiiiiiindex.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /indexxxxxxxxxxx.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.hhhhhhhhhhhtml HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.htmmmmmmmmmmm HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HHHHHHHHHHHTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HTTPPPPPPPPPPP/1.1\r\nHost: test.com\r\n\r\n

GET /index.html HTTP/11111111111.1\r\nHost: test.com\r\n\r\

Błąd dostawy

Obecny sprzęt do wstrzykiwania usterek sprzętowych dostarcza usterki poprzez zmianę poziomów napięcia, wstrzykiwanie danych za pomocą pinów testowych, a nawet impulsy EMI. Błędy mogą być dostarczane do oprogramowania za pośrednictwem dowolnego nośnika, z którego aplikacja przyjmuje dane wejściowe. W systemie operacyjnym Windows może to odbywać się za pośrednictwem systemu plików, rejestru, zmiennych środowiskowych, komunikatów systemu Windows, portów LPC, RPC, pamięci współdzielonej, argumentów wiersza poleceń lub danych wejściowych sieci, a także wielu innych mediów. Najważniejszymi mediami komunikacyjnymi używanymi przez dzisiejsze aplikacje są protokoły sieciowe TCP/IP. Korzystając z tych protokołów, możemy komunikować się z podatnymi na ataki produktami z odległych zakątków świata. W tej sekcji omówimy niektóre metody i wytyczne dotyczące dostarczania błędów przez protokoły sieciowe. Dostawa jest inicjowana z silnika modyfikacji. W każdej iteracji naszego silnika modyfikacji dostarczymy zmodyfikowane dane wejściowe do naszej aplikacji docelowej za pomocą zestawu funkcji sieciowych zaprojektowanych do dostarczania danych przez połączenia TCP/IP. Po zmodyfikowaniu naszych danych wejściowych dostarczymy zmodyfikowane dane w następujący sposób:

1. Utwórz połączenie sieciowe z aplikacją docelową.

2. Wyślij nasze zmodyfikowane dane wejściowe przez utworzone połączenie.

3. Poczekaj chwilę na odpowiedź.

4. Zamknij połączenie sieciowe.

Algorytm Nagela

Algorytm Nagela, domyślnie włączony w stosie IP systemu Windows, opóźni transmisję mniejszych datagramów do czasu, aż zostaną opóźnione na tyle, aby można je było zgrupować. Ponieważ nasze testy są tworzone, dostarczane i monitorowane jako oddzielne jednostki, chcemy to wyłączyć, ustawiając flagę NO_DELAY.

Wyczucie czasu

Kwestia czasu jest trudna do rozwiązania. Wiele osób wybrałoby bardzo elastyczny czas, aby serwer mógł odpowiedzieć. Inni mogą zdecydować się na przycięcie czasu, aby skrócić czas testu. RIOT jest skonfigurowany gdzieś pomiędzy. Zaleca się konfigurowanie czasu tak, aby najlepiej pasował do oprogramowania, które wybierzesz do audytu. Aplikacje serwerowe, które nie będą odpowiadać, chyba że wyślesz prawidłowe dane wejściowe, najprawdopodobniej powinny mieć bardzo krótki czas oczekiwania. Aplikacje serwerowe, które zawsze odpowiadają, niezależnie od typu żądania, powinny mieć wyższy limit czasu. Optymalnym rozwiązaniem byłoby napisanie własnego algorytmu taktowania, który dynamicznie konfiguruje taktowanie podczas inicjowania audytu.

Heurystyka

Zawsze byliśmy fanami oprogramowania, które może dostosować się do sytuacji. Chociaż podstawowa heurystyka jest daleka od prawdziwej sztucznej inteligencji, jest to interesujący krok we właściwym kierunku i zapewnia dodatkową przewagę nad wstrzykiwaniem błędów. Heurystyka to nauka o komunikowaniu się i obserwowaniu odpowiedzi w celu edukowania komunikatora. Jeśli chcesz włączyć prostą heurystykę do swojego pakietu do wstrzykiwania błędów, po prostu dodaj obsługę oddzwaniania bezpośrednio po odebranej części kodu dostawy. Możesz zacząć od sprawdzenia odpowiedzi serwera pod kątem niestandardowych kodów błędów. Gdy aplikacja inspekcyjna otrzyma błąd, taki jak wewnętrzny błąd serwera, można ustawić flagę, aby inspekcja tymczasowo stawała się bardziej agresywna, dopóki odpowiedź nie zmieni się z powrotem. Chociaż tego typu błędy serwera

sieci Web mogą wystąpić z powodu złej konfiguracji lub niepowodzenia w zainicjowaniu funkcji, mogą również wystąpić z powodu uszkodzenia przestrzeni adresowej procesu.

Protokoły bezstanowe a protokoły stanowe

Protokoły możemy podzielić na dwie klasy - protokoły bezstanowe i protokoły oparte na stanach. Protokoły bezstanowe są bardzo łatwe do audytu — wszystko, co robimy, to przesyłanie danych o błędach do aplikacji zdalnego serwera i monitorowanie jej zachowania. Protokoły oparte na stanach są nieco trudniejsze do audytu. Niewiele, jeśli w ogóle, rozwiązania do wstrzykiwania błędów oferują możliwość audytu złożonych protokołów opartych na stanie. Problem ten wynika ze złożoności negocjacji protokołu. Produkty programowe często zawierają złożone protokoły klient-serwer, które wymagają negocjacji tak szczegółowych, że prosta analiza logiczna nie może ich odtworzyć. Niewielu badaczy było w stanie opracować skuteczne systemy audytu oparte na stanach, które działają wyłącznie na logicznej analizie danych protokołu. Jedyne rozwiązania tego problemu wymagają dodatkowego kodu i/lub złożonych końcówek protokołu, które definiują każdy stan protokołu.

Monitorowanie usterek

Monitorowanie usterek, krok, który często jest rażąco pomijany, jest kluczową częścią testowania usterek. Większość projektów wstrzykiwania błędów opracowanych przez społeczność akademicką wykrywa awarie aplikacji tylko wtedy, gdy ulega ona awarii lub zrzuca jej rdzeń. Aplikacje korporacyjne są prawie zawsze budowane z silną odpornością na błędy przy użyciu obsługi wyjątków, obsługi sygnałów lub dowolnej innej obsługi błędów dostępnej z nadrzędnego systemu operacyjnego. Monitorując nasze błędy za pomocą podsystemu debugowania systemu operacyjnego, możemy wykryć wiele błędów, które wcześniej zostały przeoczone.

Korzystanie z debugera

Jeśli interaktywnie testujesz błędy, debuger będzie odpowiadał Twoim potrzebom. Wybierz debugger i dołącz proces audytowanego produktu. Wiele debugerów jest domyślnie skonfigurowanych do przechwytywania tylko wyjątków, które nie są obsługiwane przez proces; na przykład nieobsłużone wyjątki. Inne debugery umożliwiają przechwytywanie tylko nieobsłużonych wyjątków. Jeśli debuger jest w stanie wyłapać wyjątki, zanim zostaną one przekazane do aplikacji „pierwszej szansy", zalecamy włączenie tej funkcji dla każdego typu wyjątku które chcesz monitorować. Najważniejszymi wyjątkami do monitorowania są wyjątki dotyczące naruszeń dostępu. Naruszenia dostępu są generowane, gdy wątek w procesie próbuje uzyskać dostęp do adresu, który nie jest prawidłowy w przestrzeni adresowej procesu. Naruszenia te są często obserwowane, gdy struktury danych wyznaczone do pamięci odniesienia ulegają uszkodzeniu podczas działania programu.

FaultMon

Niestety bardzo niewiele debuggerów pozwala na rejestrowanie wyjątków i automatyczne kontynuowanie operacji. Z tego powodu udostępniliśmy FaultMon, narzędzie napisane przez Dereka Soedera, członka grupy badawczej eEy. Aby użyć FaultMon, po prostu otwórz wiersz poleceń i wprowadź identyfikator procesu dla aplikacji, dla której chcesz monitorować wyjątki. Za każdym razem, gdy generowany jest wyjątek, FaultMon wyświetli na konsoli informację o wyjątku.

21:29:44.985 pid=0590 tid=0714 EXCEPTION (first-chance)

Exception C0000005 (ACCESS_VIOLATION writing [0FF02C4D])

```
----------------------------------------------------------------
EAX=00EFEB48: 48 00 00 00 00 00 F0 00-00 D0 EF 00 00 00 00 00

EBX=00EFF094: 41 00 41 00 41 00 41 00-02 00 41 00 41 00 41 00

ECX=00410041: 00 00 00 A8 05 41 00 0F-00 00 00 F8 FF FF FF 50

EDX=77F8A896: 8B 4C 24 04 F7 41 04 06-00 00 00 B8 01 00 00 00

ESP=00EFEAB0: 38 25 F9 77 70 EB EF 00-94 F0 EF 00 8C EB EF 00

EBP=00EFEAD0: 58 EB EF 00 89 AF F8 77-70 EB EF 00 94 F0 EF 00

ESI=00EFEB70: 05 00 00 C0 00 00 00 00-00 00 00 00 B4 69 CC 68

EDI=00000001: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??

EIP=00410043: 00 A8 05 41 00 0F 00 00-00 F8 FF FF FF 50 00 41

--> ADD [EAX+0F004105],CH
----------------------------------------------------------------
Continue? y/n:
```

Tutaj widzimy przykładowy wyjątek, który został przechwycony przez FaultMon podczas testu RIOT. Opcja interaktywna została ustawiona na -i. Mając ustawioną opcję interaktywną, możemy robić przerwy między wyjątkami i badać stan programu.