Česká společnost uživatelů otevřených systémů EurOpen.CZ
Czech Open System Users' Group
www.europen.cz

49. konference

Sborník příspěvků

Hotel Kurdějov
9.–12. 10. 2016

Programový výbor:

Vašek Matyáš	Zdeněk Říha
Petr Hanáček	Andriy Stetsko
Jan Krhovják	Jan Hajný
Marek Kumpošt	Petr Švenda
Václav Lorenc	Nikos Mavrogiannopoulos

Sborník příspěvků z 49. konference EurOpen.CZ, 9.–12. 10. 2016

Plzeň 2016. První vydání.
Editor: Petr Švenda
Sazba a grafická úprava: Ing. Miloš Brejcha – Vydavatelský servis, Plzeň
e-mail: servis@vydavatelskyservis.cz

Tisk: Typos, tiskařské závody, s. r. o.
Podnikatelská 1160/14, Plzeň

Příspěvky neprošly redakční ani jazykovou úpravou.

ISBN 978-80-86583-31-

Obsah

Zpracování dat pomocí nástrojů Splunk a Elastic Stack

Vít Bukač, Václav Lorenc

E-mail: bukac@hotmail.cz

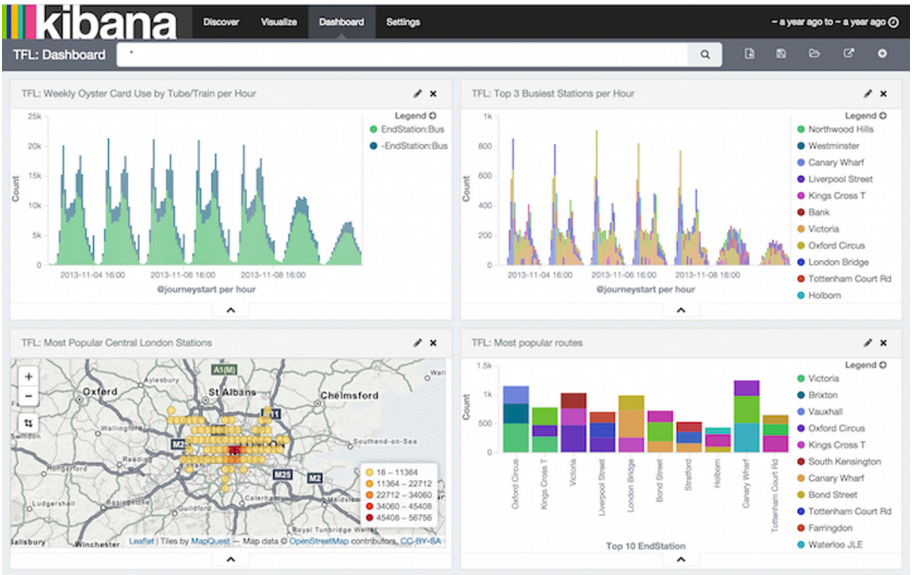
"Splunk is the new Excel!"

Není tomu tak dávno, co se ke zpracování mnoha a mnoha dat nezřídka používal Microsoft Excel. Ještě několik let nazpět bylo možné potkat bezpečnostní experty převádějící celou řadu forenzních nálezů a logů do CSV souborů. Na internetu se daly najít dobře dokumentované postupy využívající pokročilé vlastnosti Excelu. Jeho schopnost rychlého a interaktivního filtrování dat, jednoduchost ovládání i převodu dat do formátu, který umí zpracovat, v kombinaci s vytvářením nejrůznějších grafů, dlouho nenalézala přemožitele. A ačkoliv pro menší analýzy stále může být nástrojem více než vhodným, některá jeho omezení vedla k hledání dalších řešení.

Obr. 1: Ukázka vizualizace dat v prostředí Splunk

V tutoriálu si představíme nástroje Splunk a Elastic Stack, jejichž použití může být stejně snadné nebo snazší než u Excelu, jejich schopnosti zpracování velkých dat ovšem mnohem větší. A nejde jen o data velká – oba nástroje umí, na rozdíl od Excelu, výborně zpracovávat data získávaná v reálném čase z jiných systémů (např. Twitter), obohacovat je o další informace, provádět nad nimi netriviální dotazy (do jisté míry podobné SQL dotazům nad tradičními databázemi), využívat paralelních dotazů pro násobně rychlejší zpracování dat, vytvářet jejich grafové vizualizace a nad nimi i dashboardy a v neposlední řadě nastavovat nad zpracovanými daty notifikace.

Splunk je komerční řešení pro zpracování velkých objemů více či méně strukturovaných dat, Elastic Stack (dříve a častěji známý jako ELK, kombinace řešení Elasticsearch, Logstash a Kibana) je open source varianta téhož.



Obr. 2: Ukázka vizualizace dat v prostředí Kibana4

Čemu se tutoriál nebude věnovat je škálování těchto systémů pro opravdu velké datové objemy, neboť s tím nemá ani jeden z autorů žádné zkušenosti. Naopak mají zkušenosti s využitím těchto systémů pro praktické každodenní aplikace v oblasti bezpečnosti či monitoringu dat.

K čemu se tedy takové systémy dají využít právě ve vaší organizaci a pro zlepšení bezpečnosti?

Dle zkušenosti autorů stále existují týmy, které pro zpracování logů používají své ručně psané programy vzniklé před mnoha lety. Ačkoliv dostatečně fungují

a reporty z nich stále chodí, jejich udržování, opravy a přizpůsobování novým zařízením i novým typům dat vyžadují nemalé úsilí.

Zkusíme si proto nastavit vlastní Splunk a ukázat si, jakým způsobem je možné do něj dostat požadovaná data, jak vypadá dotazovací jazyk a jaké možnosti vizualizace poskytuje. Totéž se poté pokusíme do nějaké míry replikovat i nad ElasticStackem, ukážeme si rozdíly i společné vlastnosti.

Nejčastěji se obě řešení používají na zpracování dat v časových řadách, tedy právě na logy, záznamy událostí či měření daných veličin v časových intervalech. Zkusíme alespoň v rychlosti nastínit, jakým způsobem je možné posílat logy do jednoho či druhého systému tak, aby byl možné využít centralizovaného zpracování dat jejich pomocí.

Z vizualizací pak bývá velmi oblíbenou časový histogram, který ukazuje, kolik událostí se objevilo v kterém okamžiku. Krom pravidelného přehledu o tom, že všechny systémy fungují jak mají, nám taková informace může usnadnit budoucí plánování infrastruktury (např. u velikosti datových toků, počtu zpracovaných e-mailů, četnosti HTTP dotazů) či zvýraznit věci na první pohled méně viditelné (může se týkat výpadku záloh, příchozích phishingových kampaní či nadměrného provozu v nečekaných částech sítě).

Vývojáři aplikací mohou do obou řešení zasílat aplikační chyby, aplikační statistiky či audity přístupů a následně je přehledně vyhodnocovat a prioritizovat.

Právě rychlá analýza nad vybranou podmnožinou dat, různorodé datové zdroje, schopnost interaktivního filtrování a rychlého nalezení podobností či anomálií se používá nejen v monitoringu infrastruktur, ale i při vyšetřování bezpečnostních incidentů. Moderní bezpečnostní operační centra využívají mnoha datových zdrojů pro korelace dat a rychlé, automatizované přípravy podkladů pro vyšetřování incidentů, často provázané s některým ze známých systémů pro správu incidentů. Z pohledu manažerů pak systémy typu Splunk či Elastic Stack umožňují snadné vizualizace nastavených metrik, což může zrychlit rozhodovací procesy v organizaci. Tomu všemu bychom rádi věnovali alespoň část našeho tutoriálu.

Na základě zájmu a zkušeností publika se také pokusíme předvést několik dalších projektů využívajících ElasticSearch, například *nightHawk Response* [4] pro rychlejší analýzu bezpečnostních incidentů.

Pro potřeby tutoriálu budeme využívat zdarma dostupnou verzi Splunku [1] s omezením na 500 MB dat denně. Vše potřebné pro využití Elastic Stacku bude předpřipravené v obrazu pro některé z virtualizačních řešení (VMWare nebo VirtualBox), což je i největší nezbytná nutná podmínka pro praktickou část tutoriálu. A pokud se vám nechce čekat až na tutoriál nebo trávit čas instalací ať již Splunku nebo Elastic Stacku, můžete směle využít možností řešení 21. století – obě řešení (Splunk Cloud [2] či u Elastic Stacku např. ve variantě Logsene [3]) je možné zdarma za určitých podmínek využívat v cloudu. Jen si řádně rozmyslete, jaká data tam chcete či můžete posílat.

Literatura

[1] *Try Splunk Cloud For Free.* Splunk, Inc. [online]. (cit. 01. 09. 2016)
Dostupné z: https://www.splunk.com/page/sign_up/cloudtrial

[2] *Free Trials and Downloads.* Splunk, Inc. [online]. (cit. 01. 09. 2016)
Dostupné z: https://www.splunk.com/en_us/download.html

[3] *Log Management & Analytics. Sematext.* [online]. (cit. 01. 09. 2016)
Dostupné z: https://sematext.com/logsene/

[4] *nightHawk Response. Asynchronus forensic data presentation.* [online]. (cit. 15. 09. 2016)
Dostupné z: https://github.com/biggiesmallsAG/nightHawkResponse

Virtuální čipová karta Remsig

Daniel Kouřil, Michal Procházka

E-mail: kouril@ics.muni.cz

V oblasti zpracování elektronických dokumentů hrají důležitou roli digitální podpisy, které umožňují realizovat řadu bezpečnostních funkcí, jako je kontrola integrity a ověření autenticity zpráv. Podepisovací technologie, které se dnes používají v běžné praxi, jsou založeny na využití asymetrické kryptografie, kdy pro vytvoření podpisů uživatel používá svůj soukromý klíč. Podpis lze ověřit pomocí odpovídajícího veřejného klíče, který zpravidla bývá potvrzen certifikační autoritou uznávanou spolupracujícími uživateli.

V systému pro vytváření digitálních podpisů pomocí asymetrické kryptografie je nutné zajistit dostatečnou ochranu soukromého klíče tak, aby bylo omezeno riziko jeho zneužití. Jinak by potenciální útočník, který získá přístup k soukromému klíči, mohl vytvářet digitální podpisy za uživatele, aniž by je příjemce mohl odlišit od legitimně vytvořených.

Pro ochranu soukromého klíče se používají různé přístupy. Často se používají specializovaná úložiště, která jsou dostupná z uživatelské stanice a která omezují přístup ke kryptografickému materiálu. Úložiště mohou být softwarová, která jsou realizována jako aplikace nebo součásti operačního systému. Softwarová úložiště zpravidla fungují jako centralizovaný prostor pro uložení klíčů, které poskytují aplikacím. Hardwarová řešení, jako jsou čipové karty, umožňují i provádět kryptografické operace v rámci svého systému. Citlivé informace tak neopouští chráněný prostor karty a nemohou být zpřístupněny z uživatelských aplikací. Čipové karty a příbuzné technologie jsou často používanou metodou pro ochranu soukromých klíčů používaných pro digitální podpisy.

Přes velké rozšíření mají lokální úložiště (tj. úložiště spojená přímo s uživatelskou stanicí) i několik nevýhod, které omezují celkovou bezpečnost a mají vliv na celkové náklady. Ačkoliv řada řešení poskytuje dobrou úroveň ochrany, zkušenosti ukazují, že netvoří uživatelsky přívětivé prostředí. Uživatelé pak mají tendence řešení používat nesprávně, příp. je obchází úplně. Vedle ochrany soukromého klíče, je nutné dosáhnout i správné úrovně uživatelské přívětivosti tak, aby možnosti asymetrické kryptografie nebyly oslabovány nesprávným použitím.

Problémy s lokálními úložišti a lokálně vytvářenými podpisy mohou nastat v několika oblastech:

- Řízení životního cyklu úložiště. U čipových karet a jiných hardwarových řešení je nutné zajistit distribuci zařízení, jejich výměnu při poškození, odblokování apod. Tyto procesy lze poměrně snadno realizovat v prostředí s dobře ustavenou službou pro uživatelskou podporu. Problém vzniká v okamžiku, pokud taková služba k dispozici není nebo neumožňuje přímý kontakt s koncovými uživateli.
- Uživatelské návyky při používání čipových karet. Uživatelé karty používají jako prostředek pro vytváření podpisů, ale nic je nenutí k tomu je používat korektním způsobem. Např. je možné, aby uživatel nechával zapojenou kartu zapojenou ve čtečce neustále, což zvyšuje možnost zneužití, pokud se útočník má možnost dostat k systému uživatele. V organizacích, kde neexistuje koordinace pořizování hardware, může snadno docházet k problémům s kompatibilitou. Podobné problémy se mohou objevit, pokud uživatelé používají odlišné operační systémy nebo aplikace.
- Zapojení podepisování do pracovního workflow. Digitální podpisy často slouží k potvrzování dokumentů, které jsou generovány v rámci interních systémů. V dnešní době je běžný přístup přes webové portály, které zpřístupňují aplikace a vytvářené dokumenty. Integrace lokálně generovaných podpisů do těchto aplikací je složitá z pohledu uživatele i provozovatele systému. Pokud je potřeba podepsat dokument, který je uložen na vzdáleném úložišti, je nutné, aby jej uživatel zkopíroval na svůj pracovní počítač, podepsal jej a opět uložil zpět do úložiště.
- Uživatelé jsou zvyklí používat mobilní zařízení i pro zpracování elektronických dokumentů. Používání čipových karet je většinou obtížné, protože mobilní zařízení nedisponují stejnými možnostmi připojení jako osobní počítače, pro které technologie vznikala.

Jako alternativa k lokálně instalovaným úložištím, existuje řešení, které umožňuje přenést úložiště na vzdálený server a použít jej i jako prostředek pro vytváření digitálních podpisů. Služba RemSig poskytuje implementaci řešení pro vzdálené podepisování dokumentů.

1 Služba RemSig

Služba RemSig byla navržena jako řešení, které kombinuje spolehlivou ochranu soukromého klíče a vysokou úroveň uživatelské přívětivosti. RemSig vytváří službu, která spravuje soukromé klíče a umožňuje s nimi vykonávat kryptografické operace. Přístup je službě probíhá pomocí síťového rozhraní. Vytváří analogii čipové karty, která ovšem nevyžaduje fyzické zařízení spravované koncovým uživatelem.

RemSig je síťová služba, která nabízí dvě základní třídy funkcí. První je úložiště certifikátů a soukromých klíčů. RemSig je navržen tak, aby pokryl celý

životní cyklus certifikátu veřejného klíče. Umožňuje vytvořit žádost o certifikát, včetně kontrolovaného procesu generování dvojice kryptografických klíčů. Uživatel nemusí řešit technické aspekty a musí pouze projít ověřením, které závisí na požadavcích konkrétní certifikační autority.

Analogicky k čipovým kartám ponechává RemSig vygenerovaný kryptografický materiál uložen v interním úložišti. Soukromé klíče jsou uloženy na lokálním systému v souborech, které jsou zašifrovány heslem, které uživatel zvolí při vytváření soukromého klíče. Heslo je zadáváno uživatelem při provádění operací, které potřebují soukromý klíč. Během těchto operací je klíč uložen v operační paměti a je smazán v okamžiku, kdy operace skončí. Správce stroje má dostatek možností, aby mohl získat přístup do paměti a zjistit hodnoty klíčů. Na tomto místě ale poznamenejme, že výchozím předpokladem pro návrh služby RemSig bylo prostředí velké instituce s dedikovanou správou IT. Situace tedy není nepodobná správci, který má přístup k pracovním stanicím. Pokud uživatel používá čipovou kartu na těchto zařízeních, správce má přístup k operacím s kartou a může nechat vytvářet libovolné podpisy (zejména v okamžiku, kdy např. uživatelé nechávají kartu v zařízeních permanentně zapojenou).

Z důvodu napojení na existující aplikace bylo nutné umožnit uživatelům přístup k jejich podepisovacím datům tak, aby si mohli certifikát i soukromý klíč z úložiště stáhnout. RemSig proto umožňuje export podepisovacích dat ve formátu PKCS12. Tato možnost jde proti duchu uzavřeného úložiště a považujeme ji za dočasnou funkčnost, která by měla být potřeba pouze pro přesně vyhrazené účely a v budoucnu nejlépe odstraněna.

RemSig pomáhá udržovat přehled o certifikátech všech uživatelů, což lze s výhodou využít v okamžiku, kdy je např. potřeba certifikát revokovat.

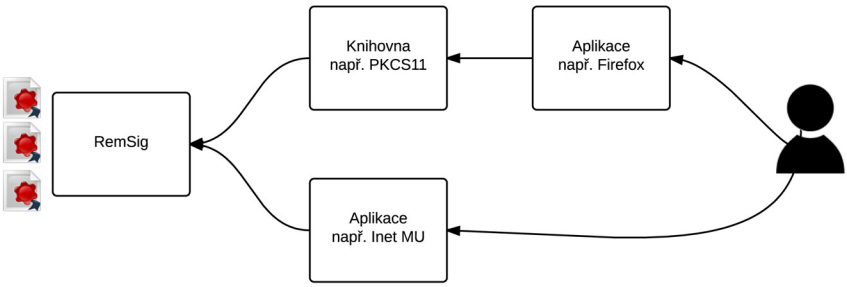
Vedle funkce úložiště nabízí RemSig operace, které pracují se soukromým klíčem tak, aby jej nebylo nutno exportovat mimo chráněný prostor. Služba nabízí rozhraní, které umožňuje vytvářet digitální podpisy zaslaných dat. Podporováno je vytváření podpisů v několika formátech.

Nejčastěji se používá podpis PDF dokumentu, kdy na vstupu RemSig obdrží PDF dokument a vrací jiný PDF dokument, který obsahuje podpis vytvořený uživatelovým soukromým klíčem. Pro vložení podpisu do PDF dokumentů se využívá vlastnosti PDF formátu, kdy lze digitální podpis vložit přímo do dokumentu. Elektronický podpis lze přímo ověřit v prohlížečích PDF formátu, které mají podporu pro digitální podpisy (např. Adobe Reader). RemSig může u PDF vložit do dokumentu vodoznak, v němž je specifikováno, kdo a kdy dokument digitálně podepsal. Další možností je získání podpisu generických dat ve formátu PKCS7. RemSigu jsou předána vstupní data a zpět se vrátí pouze podpis dat. Jelikož parametrů pro digitální podpis může být více a lze využít různé algoritmy pro výpočet otisku dokumentu nebo šifrování, podporuje RemSig tzv. podpisové profily. Při požadavku o podpis lze specifikovat profil, který je podporován systémem RemSig. V současné době jsou podporovány profily pro ČNB a ČSSZ,

které splňují konkrétní požadavky služeb těchto institucí na formát digitálních podpisů.

RemSig je serverová aplikace, která poskytuje pouze aplikační rozhraní. Veškerá interakce uživatele je realizována přes externí systémy, se kterými uživatel pracuje. V typickém scénáři se jedná o informační systémy instituce nebo služby řídící oběh elektronických dokumentů. Komunikace se službou RemSig probíhá přes protokol HTTPS a pro přenos zpráv se používá dokumentů v jazyce XML. Toto řešení bylo zvoleno z důvodů široké podpory na straně systémů a možnosti rozšiřitelnosti. Přes aplikační rozhraní je možné používat obě zmíněné třídy funkcí, tj. spravovat podepisovací data a provádět digitální podepisování.

RemSig lze využít také v koncových aplikacích, případně ve službách operačních systémů pracujících s certifikáty. V těchto případech se využívá standardizovaných rozhraní pro přístup k certifikátům. Jedním z rozhraní je standard PKCS11, který je podporován řadou aplikací, např. Firefox a Thunderbird. Pro RemSig byla vyvinuta knihovna PKCS11, která umožňuje využívat certifikáty uložené v RemSigu přímo koncovými aplikacemi. RemSig se pro aplikace tváří jako běžná čipová karta.



Obr. 1: Schéma interakce uživatele s RemSigem.

Řízení přístupu a autentizace ke službě RemSig je řešeno na dvou úrovních. K serveru RemSig se musí nejprve autentizovat aplikace nebo knihovna, přes kterou uživatel chce provádět digitální podepisování nebo manipulovat s certifikáty nebo soukromými klíči. Následně dochází k sekundární autentizaci uživatele, kdy uživatel odemyká svůj soukromý klíč heslem, které si zvolí při vkládání klíče do systému. RemSig podporuje dvě role pro systémy, které s ním komunikují. První umožňuje manipulaci s certifikáty uživatele, např. generování žádosti, import certifikátu, revokaci. Druhá role je určena jen pro systémy, které chtějí podepisovat data.

U centrálního řešení jsou vždy vyšší nároky na zabezpečení, a zároveň musí splňovat legislativní požadavky dané na zařízení a procesy při manipulaci s kvalifikovanými certifikáty. Bezpečnost u centrálního úložiště certifikátů musí být

řešena už od fyzické bezpečnosti až po nemožnost administrátora systému přistoupit k otevřené formě privátního klíče. Systém RemSig je koncipován tak, že se na fyzickém disku nikdy neobjeví citlivá data, tj. soukromé klíče, v otevřené formě. Jelikož RemSig pracuje s citlivými daty, samozřejmosti je kompletní logování a auditování. Každá operace je zaznamenána, a to včetně osoby, která ji provedla, kdy a s čím manipulovala.

Se soukromými klíči se pracuje v operační paměti stroje, na kterém běží služba RemSig.

2 Nasazení služby RemSig na MU

RemSig je v produkčním provozu na Masarykově univerzitě od roku 2014. Celkem spravuje téměř 500 certifikátů od cca 170 uživatelů z různých organizačních jednotek univerzity. Uživatelé ji používají primárně pro správu tzv. zaměstnaneckých certifikátů, ale mají možnost ji použít i jiné typy certifikátů.

RemSig je integrován s informačním systémem Inet MU, který zajišťuje přístup k osobní a ekonomické agendě. Do systému Inet byla implementována podpora pro získávání certifikátů od certifikační autority PostSignum České pošty. Pokud uživatel zadá do systému požadavek na nový certifikát, systém zajistí, že nad ním proběhne schvalovací proces v rámci univerzity, případně v rámci fakulty, kam uživatel patří. Následně se v systému RemSig nechá vytvořit dvojice klíčů. Výsledkem této operace je žádost o certifikát, která obsahuje veřejný klíč. Žádost je přes volání API předána certifikační autoritě k dalšímu zpracování. Uživatel je následně vyzván k ověření své identity, které lze provést na registračních místech certifikační autority. V případě PostSignum může uživatel za účelem ověření identity využít kteroukoli poštu nebo jiné místo se službou CzechPOINT2. Jakmile je uživatel ověřen, certifikační autorita podepíše certifikát jeho veřejného klíče a zpřístupní jej ke stažení. Systém Inet MU pravidelně kontroluje přítomnost nových certifikátů pro zaměstnance MU, které jsou automaticky stahovány. Po stažení je certifikát nahrán do systému RemSig, kde je spárován s odpovídajícím soukromým klíčem. Od tohoto okamžiku je k dispozici uživateli, který s ním může provádět podepisovací operace.

Komunikace mezi informačním systémem Inet MU a certifikační autoritou PostSignum probíhá přes standardní šifrované HTTP požadavky, zasílané metodou POST na adresu služby. Součástí požadavku vždy musí být XML soubor dle předepsaného formátu. Odpověď služby obsahuje XML zprávu obsahující výsledek operace a případně další informace. Pro plný přístup je nutné pro komunikaci systémů použít komerční serverový certifikát, v případě PostSignum vydaný právě touto autoritou.

Mezi základní funkce rozhraní patří metody pro správu žadatelů (uživatelů intranetového systému, resp. zaměstnanců univerzity) a jejich tzv. pravidel pro

vydání certifikátů. Kromě toho rozhraní poskytuje metody pro import žádosti o certifikát, přijetí (resp. potvrzení), stažení vydaného certifikátu nebo odeslání žádosti o prodloužení certifikátu. Poslední zmíněnou metodu je možné volat pouze v případě, že má uživatel stále platný certifikát, kterým žádost podepíše, a nemusí tedy již na pobočku za účelem ověření totožnosti.

V případě odcizení nebo ztráty přístupu k soukromému klíči uživatel opět navštíví aplikaci v Inetu, kde může vyvolat revokaci certifikátu. Ta má za následek zneplatnění nejen v aplikaci, ale také v RemSigu, a zadání požadavku ke zneplatnění certifikátu u certifikační autority.

Aktuálně je systém využíván aplikacemi interních i externích systémů univerzity. Mezi interní patří již zmiňovaný intranetový systém Inet MU, který plní mimo správcovské funkce také funkci odběratelskou. Kromě možnosti vzdáleného podepisování dokumentů lze např. podepisovat projektové pracovní výkazy používané např. pro evropské projekty programu H2020. Další využití je možné nalézt v centrální spisové službě, která je rovněž interním systémem univerzity, a to pro podepisování dokumentů, např. při odesílání datovou schránkou.

Dalším konzumentem je dodavatelsky řešený personální systém. Systém v rámci správy zaměstnanců komunikuje mimo jiné s Českou správou sociálního zabezpečení, které předává Evidenční list důchodového pojištění (ELDP). Podání musí být ve stanoveném formátu a data musí být elektronicky podepsána. Akce je opět realizována vzdáleně, pomocí rozhraní. Obdobně jako personální systém využívá možnosti elektronického podepisování také dodavatelsky řešený ekonomický systém. V tomto případě jde o odesílání bankovních příkazů do České národní banky (ČNB) pomocí služby internetového portálu ČNB ABO-K (internetové bankovnictví).

3 Závěr

Patřičná míra uživatelské přívětivosti je důležitý faktor, který může zásadně ovlivnit úroveň zabezpečení. Služba RemSig poskytuje alternativu k tradičním metodám pro ochranu soukromých klíčů, která poskytuje dostatečnou míru uživatelského pohodlí při zachování zabezpečení podepisovacích dat.

Password hashing reloaded

Milan Brož, Ondrej Mosnáček

E-mail: xbroz@fi.muni.cz, xmosnac@fi.muni.cz

Abstrakt

Funkce pro odvození klíče a výpočet hashů hesel jsou používány v mnoha systémech a měly by útočníkovi znesnadnit provedení útoků hrubou silou (nebo pomocí slovníku). Popíšeme si nápady, které vzešly z Password Hashing Competition, kde bylo cílem nahradit v současnosti používané algoritmy jako PBKDF2 či scrypt. Jak je reálné provedení slovníkového útoku na funkci PBKDF2 s použitím GPU (a kolik takový útok může přibližně stát) si ukážeme na příkladu útoku na heslem chráněný šifrovaný disk (LUKS).

1 Úvod

Ukládání a bezpečná práce s hesly více uživatelů se řeší od dob, kdy se objevily víceuživatelské systémy (tedy zhruba od 60 let minulého století). Hesla jsou obecně používána k autentizaci uživatele a jejich nespornou výhodou je jednoduchost uložení a použití. Zřejmou nevýhodou je, že uložení nemusí znamenat jen zapamatování, ale například papírek s heslem na monitoru či PIN napsaný na kartě.

V nepravidelných intervalech je různými firmami předvídán konec hesel a jejich náhrada „bezpečnějším“ řešením jako jsou biometrické systémy, bezpečnostní tokeny ve formě čipových karet či generátorů jednorázových hesel nebo kombinace více způsobů (vícefaktorová autentizace). Přesto, hesla stále zůstávají, a pravděpodobně ještě dlouho zůstanou, jednou ze základních technik k autentizaci uživatele.

Nejvýraznější problémy s hesly dnes mají online služby, které musí zápasit s úniky databází používaných na ověření hesla.[1] Pomineme-li nejhorší případ, že hesla uniknou v otevřené formě (a tím pádem se okamžitě objeví v již tak

[1] https://haveibeenpwned.com/PwnedWebsites

rozsáhlých slovnících,[2] hesel), úniku dalších velkých databází s hashi hesel se nevyhneme.

Řešení, která prosazují nutnost velmi komplexního hesla a jeho změnu v pravidelných intervalech často podceňují pragmatický způsob chování běžných uživatelů [1], nebo naopak spoléhají na reálnou schopnost uživatele chtít se naučit postupy k zapamatování komplexního hesla[3] nebo použití vhodného nástroje k ukládání hesel.

Lze však i v situaci, kdy použité heslo nepatří k nejsilnějším, zkomplikovat potenciálnímu útočníkovi jeho strategii tak, že vyhledávání hrubou silou (případně i s pomocí slovníku) nad uniklou autentizační databází se stane nezajímavé či nedostupné (zejména vzhledem k ceně útoku)?

2 Password hashing

Databáze pro autentizaci uživatelů obsahují výstup z funkce (a příslušná metadata), které z tajného hesla na vstupu odvodí uložený hash. Tyto funkce jsou založeny na principu, že výpočet hashe se znalostí tajného hesla je poměrně výpočetně jednoduchý, ale inverzní operace (nalezení hesla k existujícímu hashi) je naopak velmi náročné.

Obecně jsou takovéto funkce dnes používány zejména v těchto scénářích:

- *Password hashing*, tedy jednosměrná fuknce pro ověření hesla, používaná ve zmíněných víceuživatelských systémech pro autentizaci.
- *Funkce pro odvození klíče*, Password-Based Key Derivation Function (PBKDF), kde výstup funkce je použit jako klíč pro jiný kryptografický algoritmus (například pro symetrickou šifru).

  PBKDF zajistí, že z hesla, které má potenciálně malou entropii a nevhodný formát, se vygeneruje použitelný klíč o požadované délce a formátu.
- *„Proof of Work“*, tedy data obsahující „důkaz“, že na výpočet bylo vynaloženo nezanedbatelné množství výpočetního výkonu a energie.

  Důkaz může spočívat v tom, že k hashi o daném formátu (například obsahujícím konkrétní vzor) jsou hrubou silou nalezeny odpovídající vstupní parametry. Na tomto principu jsou založeny například kryptoměny.

2.1 Krátká historie

Nejprve se pro uložení hesla používala pouze hash funkce. Evidentní slabinou tohoto přístupu mimo jiné je, že stejná hesla pro různé uživatele mají viditelně stejný hash. Proto ke vstupnímu heslu byla přidána náhodná sůl (salt), která se

[2] https://wiki.skullsecurity.org/Passwords

[3] Stačí si vyhledat heslo `Tr0ub4dor&3` na Google – goo.gl/JmwJ9M a následnou diskuzi ;-)

pak ukládá společně s hashem hesla. Tento přístup však nevyřeší relativní snadnost provést slovníkový útok, neboť výpočet jedné iterace hashovacího algoritmu je s rostoucím výpočetním výkonem velmi levný. Zavedlo se tedy opakované rekurzivní iterovaní této operace a to je v řadě algoritmů použito dodnes.

2.2 PBKDF2

Například algoritmus PBKDF2 [2], který interně používá hash funkci H, lze v principu zapsat jako

$$F_H(password,\ salt) = H(H(\ldots H(password,\ salt)\ldots))$$

kde počet iterací funkce H nastavuje časovou náročnost algoritmu.

Funkce záměrně vyžaduje serializaci výpočtu. H je kryptografická hash funkce, obvykle SHA1 nebo SHA256. Největším problémem takto konstruovaných funkcí je poměrně malá (často při běhu konstantní) paměťová náročnost.

2.3 Akcelerace na hardware

Malá paměťová náročnost umožňuje poměrně levnou implementaci na speciálním hardware, obvykle označovaném jako ASIC (Application Specific Integrated Circuit) a využití masivně-paralelních systémech jako jsou GPU (Graphics Processing Unit).

V tomto ohledu lze možnosti akcelerace poměrně dobře odvodit od realizovaných ASIC systémů používaných na dolovaní kryptoměn, kde je konstrukce motivována přímým finančním ziskem. Podle [3] je energetická náročnost specifického hardware až 2^{32} hash operací na Joule, zatímco na PC jen 2^{17}. Obdobně lze využít GPU systémy.

Obecně lze cenu za ASIC zvýšit požadavkem na konstrukční cenu (algoritmy vyžadují komplexní a drahý návrh čipů). Druhou možností je zvýšení ceny při běhu, kde se náročnost algoritmu promítne na cenu za spotřebovanou energii.

Řešením problému se zdá využití funkcí, které budou vyžadovat předem specifikované množství paměti. Větší množství RAM pro paralelní výpočty je pro ASIC i GPU systémy velmi drahé a umožňuje alespoň částečně srovnat cenu za výpočet mezi ASIC, GPU a uživatelskými systémy [11].

2.4 Scrypt a memory-hard funkce

Prvním představeným algoritmem, který vyžaduje při hashování hesel velké množství paměti je scrypt [4, 5].

Scrypt zavedl pojem paměťově náročná funkce (Memory-Hard Function, MHF), což je funkce, která pro výpočet vyžaduje předem definované množství paměti. Funkci nelze bez velmi výrazné časové nebo energetické penalizace spočítat s menším množstvím paměti (Time-Memory resp. Energy-Memory tradeoff).

Jednou z variant paměťově náročné funkce je iMHF (data-independent MHF), která při přístupech do paměti (vzorech přístupů do paměti) nezávisí na vstupních datech. U takovéto funkce je mnohem jednodušší potlačit vliv vedlejších kanálů a hodí se proto na systémy, kde lze očekávat „nepřátelské“ prostředí. Pro konstrukci, kde vyžadujeme odvození klíče, je iMHF vhodnější.

2.5 Argon2 a Password Hashing Competition

Potřeba najít, analyzovat a standardizovat nový algoritmus, který přidá nejen paměťovou náročnost, ale i schopnost využití paralelního výpočtu, vedla ke kryptografické soutěži Password Hashing Competition [6] (PHC).

V soutěži se objevilo několik vylepšení existujících algoritmů, ale také několik kompletně nových konstrukcí. Vítězem se nakonec stal algoritmus Argon2 ve variantě Argon2d (MHF) a Argon2i (iMHF).

Většina algoritmů navrhovaných v rámci PHC je postavena na využití existujícího kryptografického primitiva, například hash funkce BLAKE2b (optimalizovaná 64 bit varianta). Zajímavostí je u některých kandidátů použití algoritmů s nižším počtem rund, případně využití jen specifické vlastnosti algoritmu (například pouze kompresní funkce) [13].

Obecnými koncepty je využití MHF a striktně sériové zpracování (nebo naopak možnost definovat vyžadovaný počet nezávislých vláken).

Jedním z nových konceptů je takzvaný *Client-Independent Upgrade*, což je možnost zvýšit parametry hashovací funkce bez nutnosti zadávat vstupní heslo. Jiným zajímavým konceptem je *Server-Relief*, kde algoritmus umožňuje delegovat část výpočtu na jiný, potenciálně i ne zcela důvěryhodný, server.

U mnoha algoritmů je patrná obliba pojmenovávat parametry po ingrediencích. Již jsme zmínili sůl (salt), což je veřejná náhodná hodnota, která se přidává na počátku výpočtu a zamezuje předpočítání hashů známých hesel. Pokud část soli neuložíme (algoritmus tedy musí verifikovat více variant) tato hodnota se nazývá pepř (pepper). Parametr udávající paměťovou náročnost se pak často označuje jako česnek (garlic).

Argon2 [7] má jako primární vstup heslo a sůl o variabilní délce nepřesahující $2^{32} - 1$ znaků. Minimální délka soli je 8 bytů (16 je obvyklá délka).

Sekundárními vstupy funkce jsou parametry udávající

- úroveň interní paralelizace (počet nezávislých vláken),
- vyžadovanou velikost paměti,
- počet iterací (udávající časovou náročnost) a
- a velikost výstupního tagu (klíče).

Na počátku Argon2 spočítá hash vstupu a zaplní paměť strukturovanou do matic bloků podle požadovaného stupně paralelního zpracování. Každá matice je počítána pomocí kompresní funkce založené na BLAKE2b. Indexy bloků jsou

náhodné (Argon2d) nebo pseudo-náhodné (Argon2i). Celá operace so opakuje podle požadovaného počtu iterací. Poslední krok pak aplikuje operaci XOR na sloupce matic a spočítá finální blok. Výstup je generovaný s pomoci iterativního hashování podle požadované délky.

Obecná očekávání byla, že se Argon2 stane standardem (viz návrh RFC [8]) a postupně nahradí v nových systémech PBKDF2, bcrypt a scrypt. Realita současných dní je však taková, že se začínají objevovat teoretické koncepty jak optimalizovat výpočet iMHF funkcí [9, 10]. Praktické použití těchto analýz je stále diskutováno a závěr není jasný.

3 Příklad útoku na PBKDF2 v LUKS

Jedním z příkladů využití PBKDF2 v praxi je implementace Linuxového diskového šifrování LUKS. V tomto případě je PBKDF2 použito pro odvození klíče pro dešifrování keyslotu (speciální oblast na disku, která obsahuje vlastní šifrovací klíč). Důvěrnost dat uložených na disku jsou minimálně roky, proto je potřeba parametry nastavit a aktualizovat s ohledem na zvyšování výpočetního výkonu v tomto časovém horizontu. LUKS si při formátování otestuje počet PBKDF2 iterací tak, aby odemčení zařízení trvalo na stejném systému specifikovanou dobu (základní nastavení je 2 sekundy). S použitím nového hardware se tedy počet iterací pro nová zařízení automaticky zvětšuje.

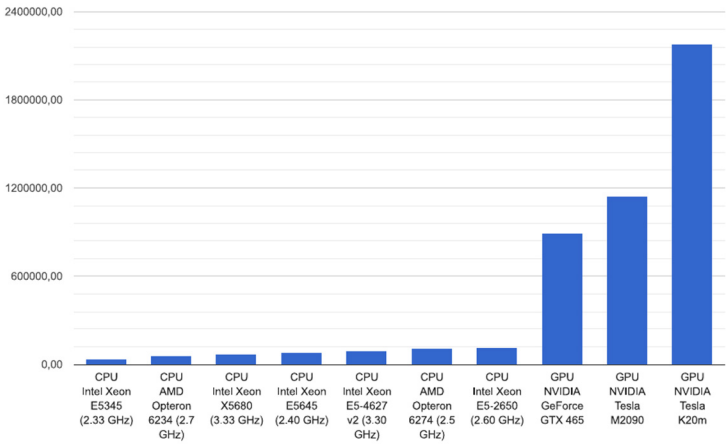
Náhrada za KDF využívající iMHF je dlouhodobou záležitostí [13]. Proto jsme se rozhodli alespoň částečně spočítat, kolik by stál útok se současnými parametry a využitím GPU.

Offline útok na LUKS na GPU je poměrně komplikovaný (vlastní dešifrování keyslotu je na GPU problematické), proto jsme na GPU implementovali pouze časově nejnáročnější část a tou je PBKDF2. Následující grafy jsou pouze ilustrační, použitý software a základní parametry vychází z prací [14, 15] a odhad ceny útoku pak z privátní konverzace (a připravovaného článku).

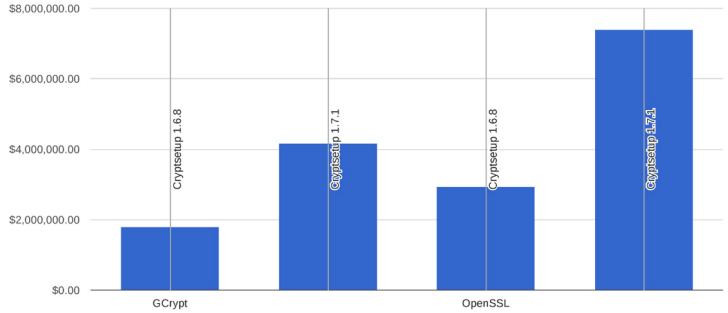
Na obrázku 1 je znázorněno, jak efektivní jsou GPU se svým paralelním zpracováním ve srovnání se současnými PC.

Obrázek 2 pak ilustruje přibližnou cenu útoku, pokud uživatel použije standardní nastavení pro LUKS. Zařízení je formátováno s použitím 256 bit klíče, s počty iterací odpovídajícím 2 sekundám pro cryptsetup 1.7 s PBKDF2-SHA256 a 1 sekundě pro crypsetup 1.6 s PBKDF2-SHA1. Cílem bylo „odhadnout“ jak změna základních nastavení mezi těmito verzemi ovlivní cenu útoku.

Průměrný změřený počet iterací pro LUKS se na současném hardware se při použití SHA1 pohybuje kolem 500 000 za sekundu (cryptsetup umí použít různé knihovny s různou optimalizací). Výpočet předpokládá maximálně 8 znakové alfanumerické heslo. V uvedeném příkladu byl použit virtuální příklad paralelního využití zařízení NVidia Tesla K20X (pořizovací hodnota $1 900 za kus, jejich



Obr. 1: Počet iterací na Joule pro PBKDF2-SHA1 [14]



Obr. 2: Přibližná cena útoku pro LUKS na NVIDIA Tesla K20X GPU

využití po dobu 15 let a ne více než 500 zařízení současně). Předpokládaná spotřeba byla 235 W při ceně $0.05 za kWh. Konkrétní algoritmus výpočtu bude součástí navazující publikace, v tomto případě jde pouze o ilustrativní srovnání ceny (reálný útok by byl asi o 20 % pomalejší kvůli použití AES pro šifrování keyslotu a anti-forenzního hashovaní keyslotu [12], které pro srovnání rychlosti PBKDF2 není relevantní).

4 Závěr

V současných systémech, které (správně) používají PBKDF2 není většinou zapotřebí žádné akutní změny. Při návrhu nových systémů je však třeba vzít do úvahy paměťově náročné algoritmy a omezit tak možnosti útočníků použít GPU nebo ASIC systémy pro útoky hrubou silou. Přestože scrypt nepatřil ke kandi-

dátům PHC (tou byl například odvozený yescrypt), RFC standard pro scrypt byl po letech dokončen a zveřejněn jako RFC 7914 [5] v srpnu 2016.

Jaký typ MHF funkce v současnosti doporučit pro budoucnost bohužel není zcela jasné. Argon2 se stále zdá být dobrým kandidátem, ale zmíněné studie ukazují, že situace není zcela jasná ani předním kryptologům.[4]

Návrh ASIC systémů je poměrně drahý a správně použitá paměťově náročná funkce jej může prodražit za hranice dostupnosti. Jak přesně drahé by byly ASIC systémy s integrovanou velkou pamětí není známo, neboť takové systémy nejsou zatím veřejně dostupné. Totéž platí pro GPU systémy, které nejsou navrženy pro výpočet funkcí s velkým množstvím aktivně využívané paměti.

Uvedený příklad ceny za útok na PBKDF2 je opravdu jen ilustrační, a je potřeba jej brát se značnou rezervou, neboť některé vstupní parametry se mohou podstatně měnit. Příkladem je cena za spotřebovanou elektrickou energii, kde v současné době lze dokonce dosáhnout v určitých oblastech a časech i negativní ceny.[5]

Literatura

[1] Cranor, L.: *Time to rethink mandatory password changes*, https://www.ftc.gov/news-events/blogs/techftc/2016/03/time-rethink-mandatory-password-changes

[2] *RFC 2898, PKCS #5: Password-Based Cryptography Specification Version 2.0*, https://www.ietf.org/rfc/rfc2898.txt

[3] *Mining hardware comparison*, https://en.bitcoin.it/wiki/Mining_hardware_comparison

[4] Percival, C.: *Stronger Key Derivation via Sequential Memory-Hard Functions*, https://www.tarsnap.com/scrypt/scrypt.pdf

[5] *RFC 7914, The scrypt Password-Based Key Derivation Function*, https://www.ietf.org/rfc/rfc7914.txt

[6] *Password Hashing Competition*, https://password-hashing.net/

[7] Biryukov, A., Khovratovich, D.: https://www.cryptolux.org/index.php/Argon2

[8] *RFC draft, The memory-hard Argon2 password hash function*, https://www.ietf.org/id/draft-irtf-cfrg-argon2-00.txt

[4] https://www.ietf.org/mail-archive/web/cfrg/current/msg08439.html

[5] https://www.cleanenergywire.org/factsheets/why-power-prices-turn-negative

[9] Alwen, J., Blocki, J.: *Efficiently Computing Data-Independent Memory-Hard Functions*, https://eprint.iacr.org/2016/115

[10] Alwen, J. et all.: *On the Memory-Hardness of Data-Independent Password-Hashing Functions*, https://eprint.iacr.org/2016/783

[11] Biryukov, A., Khovratovich, D., Groschaedl, J.: *Memory-hard functions and tradeoff cryptanalysis with applications to password hashing, cryptocurrencies, and white-box cryptography*,
https://www.cryptolux.org/images/d/d1/Tradeoff-slides.pdf

[12] *LUKS and cryptsetup*, https://gitlab.com/cryptsetup/cryptsetup

[13] Brož, M., Matyáš, V.: *Selecting a New Key Derivation Function for Disk Encryption*, STM2015.

[14] Mosnáček, O.: *Key derivation functions and their GPU implementations*, http://is.muni.cz/th/409879/fi_b/

[15] Bossi, S., Visconti, A.: *What users should know about Full Disk Encryption based on LUKS*, https://eprint.iacr.org/2016/274

Péče o knihovny třetích stran se známými zranitelnostmi

Vít Šesták

E-mail: Vit.Sestak@ysoft.com

Proč řešit knihovny třetích stran

I pokud se staráme i vlastní kód a řešíme bezpečnostní aktualizace knihoven, nesmíme zapomenout na knihovny třetích stran. Často se tyto knihovny přidávají mimo balíčky operačního systému, takže ten se o jejich aktualizace nepostará. Je to kód, který též běží jako součást naší aplikace a ovlivňuje její bezpečnost, takže to je naše starost. Ignorace této problematiky byla tak častá, ze dostala svoje místo v OWASP TOP10.

Naštěstí není vždy nutné se o každou jednu knihovnu starat ručně. Existují nástroje, které nám s nalezením zranitelných knihoven pomohou. Neudělají sice za nás veškerou práci od nalezení zranitelné knihovny přes analýzu nalezených zranitelností a aktualizaci knihovny až po vydání a distribuci opravené verze, ale aspoň tento proces zjednoduší.

Z toho ale plyne i méně příjemný důvod, proč se zranitelnostmi v knihovnách třetích stran zabývat: Tytéž nástroje má k dispozici též útočník, který se může pokoušet najít zranitelné knihovny v našem produktu, pokud má k dispozici aspoň spustitelné soubory. Zdrojové kódy nejsou vždy potřeba. Přestože má útočník méně informací, i on může tyto nástroje použít k analýze aplikace, kterou vyvíjíme. I v případě aplikace s uzavřeným zdrojovým kódem tedy může útočník získat cenné informace.

Volba nástroje pro hledání zranitelných knihoven

Zvažovali jsme více různých nástrojů podle různých kritérií. Zaměřili jsme se především na jejich praktickou použitelnost u platforem Java a .NET.

Velmi důležité kritérium je rozsah databáze. Některé nástroje používají NVD [3] nebo OSVDB [4], což jsou rozsáhlé databáze zranitelností. NVD je

k dispozici zdarma pod volnou licencí a její rozsah roste stále rychlejším tempem. OSVDB je menší, placená, roste jen přibližně konstantním tempem a do jisté míry se s NVD překrývá, přesto může mít její použití význam, zejména jako doplněk k NVD.

Další nástroje se spoléhají na vlastní databáze, což sice může znamenat přesnější výsledky (díky menšímu zapojení heuristik), ale obvykle to bylo spojeno s velmi malým rozsahem databáze. Takovéto nástroje (konkrétně SafeNuGet a Victims) jsme už dále nezkoumali. Jakkoli dobře by takový nástroj mohl dopadnout podle ostatních kritérií, těžko by mohl poskytnout relevantní výsledky.

Další kritérium je, kam putují naše data. Některé nástroje, zejména ty komerční, odesílají různé informace na server provozovatele, kde teprve probíhá analýza. Toto klade nároky nejen na důvěru v dobré úmysly, ale i na důvěru dostatečného zabezpečení serverů provozovatele, které jsou plné citlivých dat. Nasazení těchto nástrojů tak může narazit na firemní politiku, nebo se aspoň může vlivem firemní politiky zkomplikovat. Nicméně i nástroje, které analyzují na lokálním počítači mohou odesílat některá data na serveru třetích stran, když dohledávají data pro analýzu. To byl případ i nástroje OWASP Dependency Check, který ale – jak si ukážeme dále – je možné nastavit, aby tato data bral z jiných zdrojů.

Neposledním kritériem je cena. Z prozkoumaných nástrojů je v kategorii zdarma dostupných jasným vítězem OWASP Dependency Check, ostatní uvažované zdarma dostupné nástroje jsme vyřadili na základě příliš malé databáze zranitelností. V kategorii placených nástrojů je otázkou, zda za danou cenu nabídnou dostatečnou přidanou hodnotu oproti zdarma dostupnému nástroji OWASP Dependency Check. Navíc s výjimkou nástrojů z rodiny Nexus všechny uvažované komerční nástroje (tj. Veracode scanner a WhiteSource) odesílají významná data na server provozovatele, odkud si uživatel může prohlédnout výsledky analýzy.

OWASP Dependency Check

I pokud budeme dívat po komerčních nástrojích, můžeme chtít znát OWASP Dependency Check, abychom věděli, jestli má smysl připlácet za komerční řešení.

OWASP Dependency Check [1] pracuje s rozsáhlou National Vulnerability Database (NVD) [3] a podporuje širší škálu platforem – primárně platformy Java a .NET, experimentálně také Ruby, Python, Node.JS, PHP a C/C++. Je k dispozici zdarma pod licencí Apache 2.0.

Jak funguje ODC

Ve stručnosti lze fungování ODC popsat takto:

1. Projde relevantní soubory a vyhledá v nich metadata. Tato fáze záleží i na tom, který frontend používáme (CLI, pluginy pro Maven/Gradle/SBT/...), protože každý z nich v počáteční fázi hledá informace jinak.

- Najde klíčová slova v metadatech (včetně Maven Central/Nexus).
- Ke klíčovým slovům a verzi najde CPE.
- K CPE najde příslušná CVE.
- Pokud jsme nastavili potlačení některých zranitelností, zohlední je.
- Vygeneruje report ve zvoleném formátu.

Při spuštění CLI nástroje projde rekurzivně zvolený adresář a projde soubory známých typů. Pokud použijeme některý z pluginů (např. pro Maven, Gradle nebo SBT, typicky se snaží především zjistit závislosti (včetně tranzitivních) z daného projektu.

Jakkoli to může vypadat, že dostaneme výrazně rozdílné výsledky, nemusí tomu tak být. Pokud jde do analýzy soubor JAR, ODC se pokusí podle jeho otisku SHA1 dohledat jeho identifikátor v repozitáři Mavenu. Naopak, pokud použijeme například plugin pro Maven, ODC stáhne JAR soubory z repozitářů, aby v nich našel metadata. Pokud nemáme podprojekty, můžeme se tak dostat ke stejným nebo velmi podobným výsledkům při použití pluginu pro Maven jako při použití CLI nástroje nad adresářem, který obsahuje všechny závislosti projektu.

Jakmile sesbírá metadata, zkusí k nim vyhledat odpovídající identifikátor CPE („Common Platform Enumeration“) [2]. To je identifikátor softwaru (nebo teoreticky i hardwaru) používaný zejména v NVD, např. `cpe:/a:phpbb_group:phpbb-auction:1.2m`. Bohužel, tento identifikátor nelze vypočítat z metadat, dochází zde k fulltextovému vyhledávání, což je heuristika, která s sebou nese určitou chybovost. Díky tomu je ale ODC použitelný, protože si na sebe autor nevzal ambiciózní úkol udržovat mapovací tabulku mezi dostupnými identifikátory a CPE.

Jakmile jsou přiřazená CPE, je postup dále přímočarý. K nalezeným CPE se dohledají příslušné zranitelnosti v NVD, zohlední se případné potlačení některých zranitelností a výsledek se uloží ve zvoleném formátu (HTML, XML, SER).

Jak nasadit ODC

Pokud chceme skenovat projekt, musíme se nejdříve rozhodnout, který frontend použijeme. Pokud nemáme vhodný frontend, můžeme zkusit použít CLI, kterému dáme adresář se staženými knihovnami (např. JAR, DLL nebo NuSpec). Pokud

máme specifický frontend (zejména pro Maven, Gradle a SBT), můžeme použít jej. Hlavní výhodou je rozdělení reportu podle podprojektů (vyzkoušeno u Maven a Gradle).

ODC bez přístupu k internetu

Pokud chceme spouštět ODC v prostředí bez přístupu k Internetu, i to je možné. Musíme ovšem dodat potřebné informace nástroji jinak.

Zaprvé, databázi zranitelností musíme stáhnout předem. To je možné s použitím přepínače `--updateonly` u CLI nástroje. Při samotném skenu pravděpodobně budeme chtít použít volbu noupdate, aby se nepokoušel aktualizovat tuto databázi.

Zadruhé, při dohledávání JAR souborů se pokouší poslat otisk SHA1 do Maven Central. Můžeme tu nastavit URL svého serveru, podporován je i Nexus. Drobnou nevýhodou je, že se tímto můžeme připravit o některé identifikátory knihoven, které mají více různých identifikátorů.

Jak řešit falešně pozitivní výsledky

Zranitelnost můžeme potlačit dvěma způsoby. Můžeme potlačit přiřazení konkrétního souboru (podle SHA1) konkrétnímu CPE. Pokud chceme ještě jemnější pravidla, můžeme potlačit přiřazení konkrétního souboru (podle SHA1) jedné konkrétní zranitelnosti. Obojí má svoje využití.

První možnost je kolize CPE, například mrtvý pro Perlový skript Nlog i .NET knihovnu NLog se může používat CPE `cpe:/a:nlog:nlog`. V tomto případě je vhodné potlačit konktétní zranitelnosti, jinak můžeme potlačit i budoucí skutečné zranitelnosti.

Druhá možnost je evidentně špatně dohledané CPE. Podle názvu zde lze vyvodit, že CPE k dané knihovně nepatří. V tomto případě je vhodné potlačit přiřazení CPE k souboru.

Jak zpracovat výsledky z ODC

HTML report

Nejjednodušším způsobem, jak zjistit výsledky z ODC, je vygenerovat si report v podobě HTML a ten si otevřít v prohlížeči. To funguje dobře na rychlé seznámení s nástrojem nebo možná pro občasné skeny několika málo projektů.

SonarQube

K ODC existuje plugin pro SonarQube. [5] Je možné získat rychlé přehledy. Výhodou je možnost používat build breaker a snadné nasazení, pokud se již So-

narQube používá. Plugin pro SonarQube je bohužel málo vyvíjen a nepodporuje například dobře podprojekty.

ODC Analyzer

ODC Analyzer je nástroj, který jsme napsali, abychom vyřešili sledování zranitelností podle našich potřeb. Je k dispozici zdarma pod licencí BSD. [6] Ačkoli se částečně funkčně překrývá se SonarQube, oba nástroje mají jiný záběr. Tento nástroj se přímo specializuje na knihovny, umí například poradit s upgradem knihovny na novější verzi. Obsahuje různé kontroly, že skenování funguje dobře, například se kontroluje datum poslední aktualizace databáze u každého skenovaného projektu. V neposlední řadě nástroj umí export zranitelností do e-mailu nebo do systému JIRA.

Literatura

[1] *Dependency-Check – About.* https://jeremylong.github.io/DependencyCheck/

[2] *NVD – CPE Dictionary.* https://nvd.nist.gov/cpe.cfm

[3] *NVD – Home.* https://nvd.nist.gov/

[4] *OSVDB: Open Sourced Vulnerability Database.* http://osvdb.org/

[5] *Stevespringett/Dependency-Check-Sonar-Plugin.* https://github.com/stevespringett/dependency-check-sonar-plugin

[6] *Ysoftdevs/Odc-Analyzer.* https://github.com/ysoftdevs/odc-analyzer

Develop and test Web authentication with containers

Jan Pazdziora

E-mail: jpazdziora@redhat.com

1 Authentication in Web applications

Web applications that aim to be used in large enterprises and organizations need to be able to use user identities from various external identity sources within those organizations, like FreeIPA/IdM or Active Directory domains, or in case of federated setups even make use of user identities provided by completely independent entities – customer or partner organizations. Users are then authenticated via authentication methods and protocols like Generic Security Service Application Program Interface (GSS-API)/Kerberos or Security Assertion Markup Language (SAML). External identity sources can also hold and make available additional pieces of information for authorization of access to those Web applications, for instance user group membership which can map to application-specific user roles.

Many Web applications start small. Initially they might be targeted just at limited number of users so they tend to include some in-application user and user group management and handling. That is often already implemented by frameworks or development environments. For example, the Django Web framework populates its default `startproject` configuration result with `django.contrib.auth` and `django.contrib.admin`, supporting both authentication and internal user identity management features out of the box.

When external authentication is needed by first large deployment, support is quickly added for that particular protocol or mechanism, often addressing the bare minimum of features. LDAP backend support might get quickly added but without proper TLS or failover handling. Similar approach might repeat for each next protocol. Web frameworks can make this work easier but the applications developers still need to be on the lookout for new protocol requirements.

2 Offloading authentication operations

To minimize the impact that the new and evolving authentication requirements have on the development team, it is often beneficial to move the authentication operations out of the application and framework code, to a front end HTTP server, and extend the application to be able to consume the authentication and authorization results.

When Apache HTTP Server is used as the authentication front end, wealth of modules can be used for various external authentication mechanisms:

Module	**Protocol**
mod_auth_gssapi	Negotiate/GSS-API/Kerberos
mod_ssl/mod_nss	SSL client/X.509/smart-card authentication
mod_auth_mellon	SAML
mod_auth_openidc	OpenID Connect
mod_authnz_pam	Pluggable authentication modules (PAM)

The authentication for the particular application deployment can be configured in the Apache HTTP Server to use a certain module or module combination, even if the application developer team never tried that setup themselves. All that is needed is for the application to get the authentication result and act accordingly.

Of course, some rudimentary familiarity of the external protocols and their impact especially on the HTTP traffic is useful. In case of Negotiate / Kerberos authentication, HTTP responses status 401 together with `WWW-Authenticate: Negotiate` response header drives the authentication flow. For SAML or OpenID Connect, HTTP redirects to authentication providers are used. With SSL/TLS, the authentication happens (and can fail) even before the HTTP traffic is processed, while the connection is being established. When developing and testing, some external services are needed for most protocols, and for cases like Kerberos, configuration on clients might be needed as well. That configuration might however clash with the production Kerberos and other configuration used on developers' machines.

To help developers get experience with these environments and protocols, container-based Web application authentication developer setup with isolated "multihost" setup was created.

3 Web application authentication developer setup

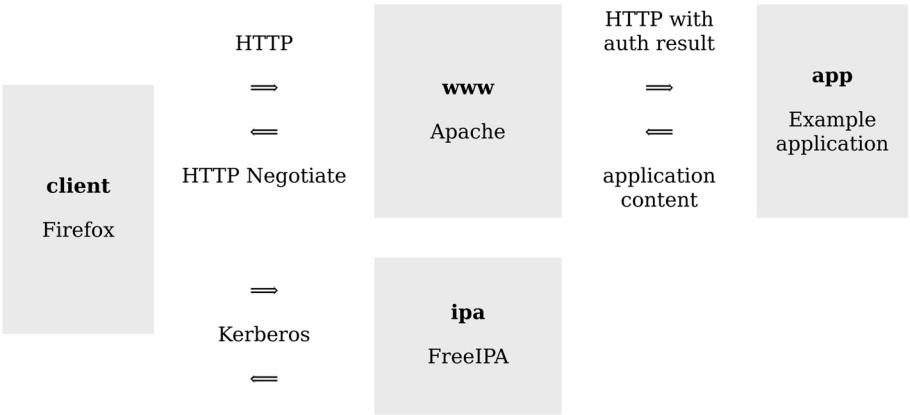
As of October 2016, the default setup consists of four containers:

Container	Purpose
ipa	FreeIPA identity management server + DNS server + Ipsilon SAML Identity Provider (IdP)
www	Apache HTTP Server authentication front-end; authentication via Apache modules
client	Firefox and Kerberos command-line tools on IPA-enrolled machine
app	Example Django application which demonstrates not just authentication behaviour but also handling of additional user attributes and use of group membership for application roles

They all run in isolated domain .example.test, managed by the integrated DNS server in the **ipa** container.

3.1 Kerberos operation

By default, Kerberos authentication via mod_auth_gssapi in HTTP authentication proxy is enabled, which leads to the following HTTP Negotiate authentication workflow:



In addition to the authentication, mod_lookup_identity with SSSD is configured in the **www** container to retrieve additional attributes about the authenticated user, like name, email address, or user group membership, and pass it to the application.

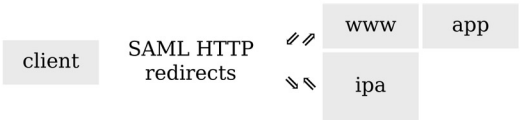
The **client** container runs SSH daemon so it is possible to start Firefox via `ssh -X`. In the same container (thus on http://localhost/ from the Firefox point of view) there is Web front-end to some Kerberos utilities: `kinit`, `klist`, `kpasswd`, `kdestroy`. That makes it easy to study the behaviour of HTTP Negotiate and Kerberos credential caches without diving into command line. That of course is also available and so is `curl`.

3.2 SAML operation

The source repository contains alternative configuration of the Apache HTTP Server authentication front-end, SAML Service Provider (SP) using mod_auth_mellon. The **www** container itself is already configured as SP for the IdP in the **ipa** container, so moving the setup from Kerberos to SAML has only two steps:

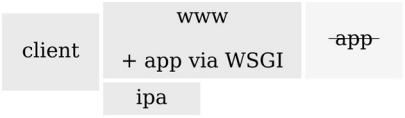
- Copy the provided Apache config file to the container data directory.
- Restart the Apache HTTP Server daemon.

Instead of HTTP Negotiate authentication and Kerberos communication, HTTP redirects for the SAML protocol will be then used:



3.3 Application on the HTTP server machine

Many Web application deployments run the application on the same machine as the HTTP end point. The developer setup supports this runtime style as well. The example application can run in the **www** container via mod_wsgi, instead of its own container via HTTP proxying:



This setup allows studying and debugging interaction of applications deployed under the Apache HTTP Server, where authentication results and additional information is passed via environment variables and similar mechanisms, instead of HTTP request headers.

As for the authentication itself, both Kerberos and SAML configurations can be used.

4 Developer setup usage

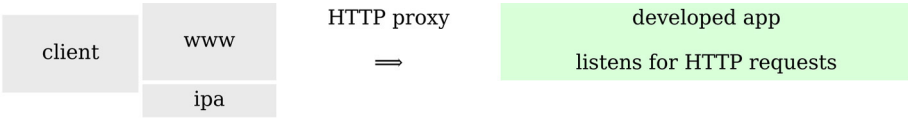
So far we have been describing the stock developer setup and its use with the example application. While that can be useful for learning about the protocol internals, developers will be more interested in ways of using the setup for their own applications that they develop. Let us look at the options. In all cases, the **app** container is no longer needed and can be disabled altogether.

4.1 HTTP proxy

Let us assume that the developed or tested application runs at its own HTTP endpoint. By merely changing the target of the default proxy directives

```
ProxyPass / http://app.example.test/
ProxyPassReverse / http://app.example.test/
```

the authentication front-end in the **www** container can be pointed to application running on the host, in different container, or on completely different machine.

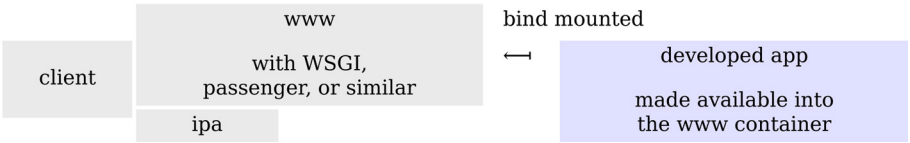


For correct operation, the configuration of the logon locations (URIs) will need to be tuned as well, to match the workflow and locations expected and supported by the application.

4.2 Application under the HTTP server

As mentioned earlier, in many deployment cases, applications will be executed by and under the HTTP server context, for instance with mod_wsgi of Apache HTTP Server. In that case, the best use of the developer setup is to run the application in the **www** container. Extending and rebuilding the setup might be needed to bring in packages of the required modules, language interpreters, or runtime environments – Ruby/passenger, Java/tomcat/AJP, etc.

The code of the developed application, potentially the working tree checkout with the source code, can then be bind-mounted to the **www** container to the location where the Apache modules will be configured to find it:

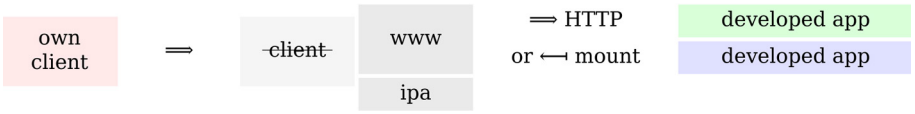


Alternatively, for example for testing, the application code can be installed to the container image during build time. Like in the previous case, the logon locations (URIs) as seen by the Apache HTTP Server and as expected by the application will need to be configured to match the application workflow.

4.3 Using own clients

The developer setup includes **client** container with Firefox and Kerberos tools installed, IPA-enrolled and part of the example.test domain. That allows for simple start, without additional configuration needed.

For development and use in continuous integration and other automated environments, other clients can of course be also used. They might need additional configuration to properly resolve hostnames of the servers in the developer setup or find the Kerberos infrastructure. For this, the integrated DNS server in the **ipa** container can be used. The **client** container automatically IPA-enrolls itself, so keytab file can be copied to those other sources.



5 Developer setup internals

As of October 2016, the containers of the setup are based on Fedora 24. Except for the **app** container, they are all systemd-based: the **ipa** container runs multiple services and both **www** and **client** IPA-enroll themselves with `ipa-client-install` which needs systemd to be run.

The first execution of the setup takes a couple of minutes (depending on the performance of the machine it is run on) because the FreeIPA server needs to be configured, using `ipa-server-install`. The configuration and data is then stored in the `ipa-data/` directory, so subsequent starts are significantly faster.

The other containers also configure themselves and their configuration and data are stored in `www-data/`, `client-data/`, and `app-data/`, respectively.

The **client** container is configured to start Firefox browser via `ssh -X`, rather than bind-mounting `/tmp/.X11-unix`. This decision was made to support more flexible and versatile, albeit potentially slower, usage.

The Web application authentication developer setup is available under the Apache License, Version 2.0.

6 Building the developer setup

The developer setup is defined as a Docker Compose application. The repository contains `docker-compose.yml` which describes the individual services – containers. Those are defined in `src/Dockerfile.*` files.

The **ipa** container's `src/Dockerfile.ipa` expects that `freeipa-server` container image is available. It is possible to build the image from sources from the

`docker-freeipa` repository, pull built images from Docker registry and tag as `freeipa-server`, or tweak the `FROM` line in the `Dockerfile` to name a specific image to use.

The container images for the setup are then built with single

```
$ docker-compose build
```

command.

7 Running the developer setup

Once the images are built, command

```
$ docker-compose up
```

will start the containers. The output will likely begin with lines similar to

```
Creating webauthinfra_ipa_1
Creating webauthinfra_app_1
Creating webauthinfra_www_1
Creating webauthinfra_client_1
Attaching to webauthinfra_app_1, webauthinfra_ipa_1, webauthinfra_www_1,
             webauthinfra_client_1
app_1     | + echo password32345
app_1     | + cp -p /var/www/django/project/db.sqlite3 /data/db
app_1     | + cd /var/www/django/project
app_1     | + python manage.py shell
app_1     | ++ cat /data/admin-password
app_1     | + echo 'from django.contrib.auth.models import User;
            User.objects.create_superuser('\''admin'\'',
            '\''admin@example.test'\'', '\''password32345'\'')'
app_1     | Python 2.7.12 (default, Aug  9 2016, 15:48:18)
app_1     | [GCC 6.1.1 20160621 (Red Hat 6.1.1-3)] on linux2
app_1     | Type "help", "copyright", "credits" or "license" for
            more information.
app_1     | (InteractiveConsole)
app_1     |
app_1     | + grep '^SECRET_KEY' project/settings.py
app_1     | + cd /var/www/django/project
app_1     | + REMOTE_USER_VAR=HTTP_X_REMOTE_USER
app_1     | + python manage.py runserver app.example.test:80
ipa_1     | Configuring ipa.example.test ...
ipa_1     | /usr/sbin/ipa-server-configure-first
www_1     | Waiting for FreeIPA server (HTTP Server) ...
client_1  | Waiting for FreeIPA server (HTTP Server) ...
```

```
ipa_1     |
ipa_1     | The log file for this installation can be found
            in /var/log/ipaserver-install.log
ipa_1     | ======================================================
ipa_1     | This program will set up the FreeIPA Server.
ipa_1     |
ipa_1     | This includes:
ipa_1     |   * Configure a stand-alone CA (dogtag) for certificate
              management
```

It shows that the **app** container quickly finishes its configuration, and **www** and **client** wait for **ipa** container to configure the FreeIPA server, to then allow them to IPA-enroll and finish their configuration.

After the startup finishes, FreeIPA server's admin password can be found in the `ipa-data/admin-password` file and private SSH key of user `developer` in the **client** container is stored in `client-data/id_rsa`. To start process in the **client** container, we find out its IP address with `docker inspect webauthinfra_client_1` and the SSH to it as user `developer`:

```
$ ssh -i client-data/id_rsa developer@172.18.0.5
```

By default the SSH daemon also listens on port 55022 on the host, so

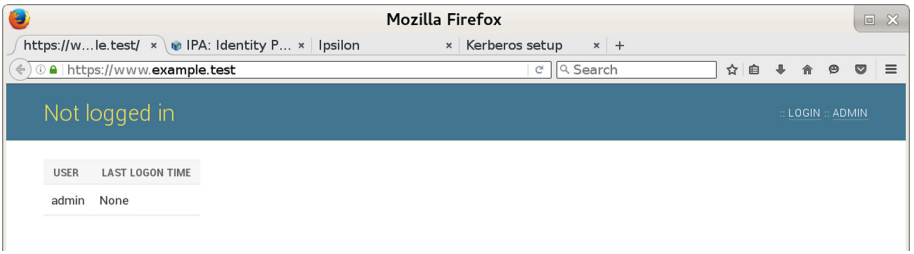
```
$ ssh -i client-data/id_rsa -p 55022 developer@localhost
```

is also possible.

To start the Firefox browser in the setup, we enable the X11 forwarding:

```
$ ssh -X -i client-data/id_rsa -p 55022 developer@localhost
      firefox -no-remote
```

The browser will open four tabs: the authenticated front-end to the example application, FreeIPA and Ipsilon IdP server logon pages, and a Web interface to Kerberos commands in the **client** container:



The Web application authentication developer setup is now ready in its default, Kerberos-enabled configuration.

8 Conclusion and future work

The container-based Web application developer setup provides an isolated environment which helps to study, develop, and test external authentication and authorization in Web applications.

In the future, configuration templates for more authentication methods might be added. We will also look at providing another example application written in different language/framework. Different software might be installed and configured in future versions to achieve the same functionality, for instance in the SAML IdP area.

Since the setup uses well-known realm, domain, and host names, it might be possible to run the initial FreeIPA server configuration (`ipa-server-install`) in build time. Unfortunately, due to the systemd-based nature and the build time environment, those investigations have not so far resulted in success. We will likely revisit that option in the future.

References

[1] *Web application authentication developer setup sources.* https://pagure.io/webauthinfra

[2] *Developer setup presentation.* https://www.adelton.com/webauthinfra/presentation/

[3] *FreeIPA container images sources.* https://github.com/adelton/docker-freeipa

[4] *Automated builds of FreeIPA container images.* https://hub.docker.com/r/adelton/freeipa-server/

[5] *Web App Authentication notes.* https://www.freeipa.org/page/Web_App_Authentication

Detekce bezpečnostních chyb pomocí statické analýzy kódu

David Formánek

E-mail: david4manek@gmail.com

Abstrakt

Práce se zabývá automatickou statickou analýzou kódu a jejím využitím pro detekci třídy bezpečnostních zranitelností nazývaných jako injekce. Kromě nezbytné teorie je především popsána vlastní implementace nového mechanismu, který pomocí tzv. taint analýzy umožňuje tyto chyby spolehlivě detekovat v Java aplikacích. Tento mechanismus byl integrován do rozšíření FindSecurityBugs pro volně dostupný nástroj FindBugs. Pokrok v analýze injekčních chyb byl ověřen na testovací sadě Juliet – přesnost detekce vzrostla na 95 % oproti 53 % u původního nástroje a rozšíření.

1 Úvod

Hlavní důraz při vývoji software je zpravidla kladen na naplnění funkčních požadavků a testována je zejména správná odpověď systému pro předepsané vstupy. Velmi důležitou otázkou je nicméně to, jak systém zareaguje na nestandardní data a okolnosti. Pokud jí není věnována dostatečná pozornost, nejenže může být narušena spolehlivost, ale jedinec s dostatečnými znalostmi může být schopen využít systém způsobem, který nebyl zamýšlen.

Klasickým bezpečnostním chybám jako *přetečení zásobníku* [1] sice brání použití jazyků s automatickou správou paměti, jiné chyby ale stejně tak mohou vést až ke kompletnímu převzetí systému, časté a přitom závažné jsou zejména tzv. *injekce*. [2] Jedná se o třídu zranitelností, kde může útočník pomocí speciálně upraveného parametru změnit logiku vykonávaného dotazu či příkazu uvnitř aplikace. Práce se zabývá nejen typy chyb, které mají slovo injekce v názvu (SQL [3], LDAP [4], XPath [5] a CRLF [6] injekce či injekce příkazu [7]), ale také dalšími problémy s obdobným principem zneužití (XSS [8], procházení adresářem [9] či neošetřené přesměrování [10]).

Protože manuální detekce chyb je velmi zdlouhavá a nákladná, i pro hledání bezpečnostních problémů je velmi výhodné použit automatické nástroje, v této práci se soustředíme na statickou analýzu [11] a jazyk Java. Pro detekci chyb typu injekce a příbuzných zranitelností se nabízí použití tzv. *taint analýzy*. Při ní jsou v kódu vyhledána potenciálně zranitelná místa (volání dotazů a příkazů) a dále je prověřována existence neošetřených cest mezi těmito místy a nedůvěryhodnými vstupy programu. Přitom není nutné analyzovaný software spouštět a chyby jsou hledány napříč celou aplikací, takže nehrozí riziko vynechání zranitelnosti v nedostatečně otestované části systému (jako při dynamické analýze).

Pravděpodobně nejpraktičtějším volně dostupným statickým analyzátorem je nástroj FindBugs [12]. Jeho zaměření na bezpečnost je však pouze okrajové a odhalena je jen malá část skutečných chyb. Naopak jeho rozšíření FindSecurityBugs [13] cílí přímo na usnadnění bezpečnostního auditu aplikací, i zde však mnoho typů vážných zranitelností zůstalo nepokryto. Velmi nízká byla také jeho schopnost rozlišovat mezi problémy skutečnými a pouze potenciálními, takže bylo třeba vždy vynaložit další úsilí na potvrzení reportovaných chyb. Cílem je proto zvýšit množství detekovaných injekčních zranitelností rozšíření FindSecurityBugs a omezit reportování volání s bezpečnými parametry. Toho je dosaženo zejména náhradou jednoduché detekce konstant vlastní implementací taint analýzy.

2 Bezpečnostní chyby typu injekce

Bezpečnostní chyby lze kategorizovat a posuzovat podle různých kritérií, takovou klasifikaci provádějí například neziskové organizace *MITRE* a *OWASP*. Obě společnosti se rovněž snaží vytipovat nejdůležitější bezpečnostní chyby, kterých by se měli (nejen) vývojáři vyvarovat. Při tom se řídí jak možnými důsledky v případě zneužití, tak i frekvencí výskytu nebo složitostí útoku. Z jejich analýz (viz dále) lze vyčíst, že automatický nástroj, jehož cílem je zvýšení bezpečnosti kódu, by se měl zaměřovat právě na chyby typu injekce. Tak můžeme označit takové typy zranitelností, kde je útočník schopen změnit příkaz (popř. dotaz či datovou strukturu) volaný uvnitř aplikace. [2] To může obecně nastat v případě, kdy je příkaz závislý na datech přicházejících od uživatele, ta nejsou dostatečně ošetřena a zamýšlený datový vstup je interpretován jako část příkazu.

Společnost MITRE mj. vytváří seznam bezpečnostních slabin zvaný *CWE (Common Weakness Enumeration)* [14], kde má každá chyba svůj číselný CWE identifikátor, popis, možné důsledky zneužití, způsoby obrany apod. MITRE ve spolupráci s institucí SANS vydala v roce 2011 seznam 25 nejnebezpečnějších chyb v software. [15] Na prvním, resp. druhém místě se nachází *SQL injekce*, resp. *injekce příkazu*. Třetí místo obsazuje klasická chyba *přetečení zásobníku (buffer overflow)*, která ale není relevantní pro vysokoúrovňové jazyky s auto-

matickou správou paměti jako je Java. Na čtvrtém místě je *XSS* a v seznamu se nachází také *procházení adresářem* a *neošetřené přesměrování*. Některé nebezpečné chyby nelze spolehlivě identifikovat automaticky – například na pátém a šestém místě jsou *chybějící autentizace* a *autorizace*, pro jejich detekci by ale automatický nástroj potřeboval informaci o tom, která data a operace jsou citlivé a jakým způsobem má být přístup k nim omezen.

Projekt OWASP (*Open Web Application Security Project*) se zaměřuje na bezpečnost webových aplikací. Publikuje návody pro jejich vývoj a testování, vytváří vlastní volně dostupné nástroje a vydává (naposledy 2013) seznam deseti nejnebezpečnějších chyb. [16] Pro každou z nich uvádí možnosti detekce a prevence, ukázku útoku, kategorizaci a další reference. Na prvním místě jsou obecně injekce (jako jedna chyba) – kromě SQL injekce a injekce příkazu jsou jsem zařazeny také *LDAP injekce* a *XPath injekce*. XSS je zařazeno zvlášť na třetím místě a neošetřené přesměrování je na místě desátém. Ostatní chyby jsou spíše generické a většinou obtížně automaticky detekovatelné.

2.1 SQL injekce

Zřejmě nejznámější a zároveň nejnebezpečnější chybou je SQL injekce. [3] Na místo očekávaného vstupu lze vložit řetězce jako `‘ OR "=‘` nebo znaky pro komentář (`#` či `--`) a tím změnit zamýšlenou logiku vykonávaného dotazu. Vytvořením vždy pravdivých nebo naopak nepravdivých podmínek v dotazu a případným zakomentováním zbytku dotazu lze například docílit toho, že při přihlášení bude z tabulky vybrán záznam daného uživatele, ale heslo nebude ověřeno. Nejenže lze výše uvedenými způsoby obejít mechanismus autentizace, ale pomocí konstrukcí s výrazy `UNION SELECT` nebo `EXISTS` a operátoru `LIKE` či funkcí dostupných v daném dialektu jazyka SQL je možné vyextrahovat obsah tabulky uživatelů i celé databáze. To lze pomocí techniky *blind SQL injection* [17] množstvím dotazů učinit pouze na základě binární odpovědi systému (úspěšná a neúspěšná autentizace) a to dokonce i pokud z odpovědi serveru není vidět, kolik měl uvnitř volaný dotaz výsledků – využívá se časově náročných funkcí v dotazech a následné analýzy doby odpovědi serveru.

V případě zneužití aktualizačních dotazů lze data také měnit či mazat a v závislosti na systému je někdy možné zcela převzít kontrolu nad serverem, kde je webová aplikace spuštěna. Konkrétní postupy závisí na použitých technologiích a jejich popis je nad rámec této práce, s využitím automatických nástrojů jako např. *sqlmap* [18] lze ale chyby zneužít s relativně malým množstvím úsilí a znalostí.

2.2 Další typy injekcí

Některé aplikace pro svou funkcionalitu vyžadují volání externích příkazů (např. příkazový řádek operačního systému). Pokud je součástí volání také parametr

pocházející od uživatele a není dostatečně validován, může dojít ke zneužití pomocí tzv. *Command injection*. [7] V důsledku může být zavolán jiný příkaz, než očekával autor aplikace, případně může útočník jako data vložit znak oddělovače (např. `;` nebo `&&`) následovaný dalším příkazem.

XSS neboli *Cross-site scripting*[1] je významná a specifická chyba, kterou i OWASP vyčleňuje mimo injekční zranitelnosti. Svým charakterem ale odpovídá injekci pro jazyk *HTML*[2], kde interpretem je přímo internetový prohlížeč návštěvníka zranitelné webové aplikace. [8] Problém nastane, pokud v sobě dynamicky generovaná stránka obsahuje neošetřený vstup od uživatele. Útočník tak může do stránky vložit nejen data, ale i HTML kód a pro následné zneužití klientský škodlivý kód (skript), obvykle v jazyce JavaScript. Přímo ohrožena není samotná aplikace, ale uživatel, který útočníkem změněnou stránku navštíví. Vložením útočníkova skriptu do stránky dojde k porušení tzv. *same-origin policy* prohlížeče a útočník získá přístup ke všem zdrojům, ke kterým může přistupovat původní stránka a její uživatelé. Lze tak libovolně číst a měnit obsah stránky (a např. vylákat citlivé údaje), vykonávat akce jménem přihlášeného uživatele, číst tzv. *cookies* (pokud nemají příznak *HttpOnly*) či podnikat další útoky z prohlížeče uživatele.

Uvažme aplikaci, která umožňuje přístup k souborům v předem daném adresáři. Pokud je cesta k souboru vytvářena zřetězením cesty k adresáři a neošetřeným názvem souboru od uživatele, může útočník pomocí sekvencí `../` přejít do nadřazených adresářů a pak uvést cestu k libovolnému souboru. [9] Tímto způsobem se lze dostat k souborům, ke kterým by normálně uživatel neměl přístup, např. zřetězení cesty `/data/public/` a vstupu `../../etc/passwd` se vyhodnotí jako `/etc/passwd` a může být přečten (popř. modifikován) soubor s přístupovými údaji.

LDAP[3] je protokol pro síťový přístup k adresářovému serveru (kde je spuštěna adresářová služba, např. *Microsoft Active Directory*). Pokud požadavek na server obsahuje neošetřený uživatelský vstup, útočník je schopen měnit LDAP dotazy podobně jako SQL dotazy u SQL injekce. Může tak obejít autentizaci, dostat se k citlivým informacím nebo i jejich změně. [4]

Jazyk *XPath* umožňuje snadný výběr (vyhledání) dat v *XML*[4] dokumentech, např. výraz `//uzly/uzel[@atr="x"]/text()` vrátí textový obsah všech uzlů s názvem `uzel`, které se nacházejí v XML tagu `uzly` a zároveň mají atribut `atr` s hodnotou `x`. Obsahuje-li XPath dotaz neošetřená data pocházející od uživatele (např. místo konstanty `x` je použita hodnota z formulářového pole), útočník může modifikovat dotaz a získat přístup k jiné než zamýšlené části dokumentu nebo změnit aplikační logiku. [5] Pokud je např. pro ověření přístupu využita XML

[1] Někdy též *CSS*, lze přeložit jako *skriptování napříč stránkou*

[2] *HyperText Markup Language*, značkovací jazyk pro tvorbu webových stránek

[3] *Lightweight Directory Access Protocol*

[4] *Extensible Markup Language*, obecný značkovací jazyk pro výměnu dat

databáze s uživatelskými jmény a hesly, lze využít operátor `or` a vždy pravdivého výrazu pro obejití autentizace podobně jako u SQL injekce.

Jako oddělovač řádků se v závislosti na platformě využívají kontrolní znaky `CR` (*carriage return,* `\r`), `LF` (*line feed,* `\n`), nebo kombinace obou. Pokud aplikace zapisuje data s neošetřenými částmi od uživatele do řádku, může jí útočník předat znaky `CR` nebo `LF` a zbytek řetězce bude interpretován jako nový řádek. [6] To může být zneužito pro falšování záznamů logu [19] nebo útoku známého jako *HTTP response splitting* [20], kdy je HTTP odpověď serveru rozdělena na dvě samostatné a oběti může být podstrčen jiný obsah.

Webové aplikace také často obsahují přesměrování na jinou stránku, pokud je cíl přesměrován na libovolnou absolutní adresu danou parametrem, jedná se o chybu *neošetřeného přesměrování (Open redirect).* [10] Spíše než o injekci se jedná přímo o použití nedůvěryhodného parametru jako adresy. To sice nepředstavuje přímé riziko pro aplikaci, ale důvěryhodnost webu je zneužita pro usnadnění útoku na uživatele. OWASP do této kategorie chyb řadí i tzv. neošetřený *forward* [21] – zde parametr obsahuje naopak název lokální podstránky (např. pro technologii JSP), která je použita pro vygenerování aktuální stránky (bez přesměrování). Chyba nastává, pokud je umožněno použít i podstránku, ke které by aktuálně přihlášený uživatel neměl mít přístup, jedná se tedy o chybu autorizace.

3 Taint analýza

Pro detekci dříve zmíněných typů chyb lze využít tzv. *taint analýzu.* Každá prakticky použitelná aplikace načítá při svém běhu nějaké vstupy. Pokud je takový vstup závislý na hodnotě od uživatele nebo komponentě, jejíž výstupy mohou být nepředvídatelné, budeme ho označovat jako *taint zdroj*[5]. Naopak bezpečné jsou konstanty uvnitř aplikace a z našeho pohledu např. také numerické hodnoty (s libovolným původem) převedené na řetězce. Problémy s neošetřenými uživatelskými vstupy se ve skutečnosti vztahují i na všechny neošetřené taint zdroje. Mezi ty patří nejen vstupy skutečně vyžádané od uživatele (pole formulářů apod.), ale také vstupy předvyplněné aplikací (např. parametry v odkazu) či data z externích komponent či systémů, jejichž validitu nemůžeme garantovat.

Aby mohl být vstup považován za ošetřený, musí být buď ověřeno, že odpovídá požadovanému formátu (znemožňujícímu útok), nebo je nutné provést konverzi, která zaručí, že speciální symboly ztratí svůj řídící význam a budou interpretovány jako běžné znaky. Protože implementace správné konverze může být náchylná na chyby (zvláště pro XSS), je vhodné spoléhat na existující řešení

[5] Anglicky *taint source*, slovo taint lze přeložit např. jako *poskvrnění* či *nákaza*, v této práci ale zachováme originální terminologii

jako ESAPI [22] a OWASP Java Encoder [23] či sanitizační metody knihoven Apache Commons Lang [24] nebo Spring [25].

Jedním ze způsobů, jak testovat přítomnost injekčních zranitelností v aplikaci, je identifikovat veškeré vstupy, posílat do nich předem připravené řetězce obsahující speciální znaky [26] a snažit se rozpoznat výstup se známkami projevené zranitelnosti. Pokud ale máme přístup ke zdrojovému kódu nebo dokážeme využít i přeložený tvar aplikace, lze se naopak zaměřit na potenciálně nebezpečné metody a analyzovat, s jakými parametry mohou být volány. Tato volání, resp. pouze jejich potenciálně nebezpečné parametry, budeme označovat jako *taint sink*[6]. V těchto místech nemusí být použit pouze taint zdroj nebo bezpečný zdroj, ale také data, která vznikla kombinací a transformací různých zdrojů. Úkolem *taint analýzy* je rozeznat, pro který taint sink existuje možnost zavolání s neošetřeným parametrem závislým na některém taint zdroji a který taint sink je naopak bezpečný. [11]

3.1 Automatická statická analýza

Manuální kontrola kódu a ruční testování běžící aplikace mohou být nezastupitelné, bez využití automatizovaných nástrojů by však náklady na detailní analýzu rozsáhlého software byly enormní. Automatizace testování umožňuje s nižšími náklady výrazně rychleji prověřit celou aplikaci, navíc způsobem, který může být snadno deterministicky zopakován a použit k regresnímu testování či kontrole kvality a analýze trendů.

Při dynamické bezpečnostní analýze musí být obvykle sestavena a spuštěna kompletní aplikace, které jsou tradičně zasílány předem připravené vstupy a zkoumány jsou pouze výstupy. Výhodou tohoto přístupu je především jeho reálnost, detekce jsou obvykle skutečné problémy a problematické vstupy lze snadno ověřit. Naopak dynamickým analyzátorům často unikají chyby, které nastávají pouze za velmi specifických podmínek, o to ale mohou být nebezpečnější. Jinou formu dynamické analýzy je tzv. *taint mode* programovacího jazyka Perl, který provádí taint analýzu za běhu a určitá volání neumožní vykonat, pokud není taint zdroj validován, přestože bezpečný běh není zcela garantován. [27]

V této práci se nadále budeme zabývat pouze statickou analýzou, při které nedochází ke skutečnému spuštění kódu, analyzátor nahlíží dovnitř aplikace a pokouší se nalézt problematická místa vyhledáváním určitých vzorů, případně abstraktní simulací vykonávání kódu. Největší výhodou oproti dynamickému testování je teoretické pokrytí celého analyzovaného kódu, aniž by bylo nutné hledat rozmanité vstupy, vyhodnocen je celý program. Naopak není nutné, aby byla celá aplikace funkční, analýza může být prováděna už od začátku vývoje a chyby mohou být nalezeny i v neúplných částech. Navíc je nástrojem obvykle přímo

[6] *Sink* lze přeložit jako *stok* či *dno*, nadále ale budeme používat originální termín

označeno místo chyby a popis nebezpečí, vývojáři je tak poskytována i určitá forma vzdělání. Kvůli tomu, že aplikace není doopravdy spuštěna, ale mohou analyzátoru chybět informace o tom, jakým způsobem bude software využíván a s jakými technologiemi použit, a ne všechny detekce jsou relevantní pro daný kontext.

3.2 Efektivní analýza

Nejjednodušším způsobem detekce potenciálních injekčních zranitelností je označení všech volání známých jako taint sink. I taková analýza může značně urychlit manuální kontrolu na dané chyby, protože omezí množství kódu, které je nutné projít, od automatického nástroje ale očekáváme schopnost rozlišení mezi bezpečným a zneužitelným voláním. K tomu je potřeba nejen detekovat a klasifikovat taint zdroje, ale hlavně analyzovat možné toky dat v aplikaci a také označit data, u nichž byla provedena konverze.

Pro statickou analýzu programů již bylo vyvinuto množství pokročilých metod jako symbolické spouštění (symbolic execution) [28], deduktivní verifikace [29], model checking [30] či abstraktní interpretace [31]. Tyto formální metody umožňují verifikovat vlastnosti systému s jistotou, popř. za daných podmínek (model odpovídá skutečnosti, chyba se projeví do k kroků apod.), nástroje však obvykle nejsou jednoduché na použití a vyžadují správnou konfiguraci pro daný systém. Hlavním problémem je ale často rychlý nárůst času a paměti potřebných pro analýzu rozsáhlejších systémů a tedy praktická neschopnost verifikace běžných programů.

My naopak budeme vyžadovat snadné použití s minimem konfigurace a hlavně škálovatelnost, která umožní analyzovat reálné aplikace (např. se stovkami tisíc řádků kódu). Formální metody mohou sloužit jako inspirace (zejména abstraktní interpretace), je ale nutné se vzdát jejich přesnosti a garance. Kvůli aproximacím nemusí být všechny chyby nalezeny a reportované problémy naopak nemusí být skutečné. Při analýze toku dat pro účely taint analýzy budeme abstrahovat hodnoty do stavu *tainted* reprezentujícího potenciálně nebezpečné řetězce a *safe* reprezentujícím bezpečné vstupy. Pokud pro danou hodnotu nemůžeme rozhodnout, zda může být bezpečná, nebo ne, můžeme ji považovat za bezpečnou za cenu neodhalení všech chyb, nebo za nebezpečnou za cenu falešných detekcí, lze ale také použít nový stav *unknown* a toto rozhodnutí tak odložit na dobu reportování výsledků.

Důležitou strukturou (používanou i formálními metodami) je tzv. *graf řízení toku (control flow graph)*. Je to reprezentace programu (v našem případě pouze jedné metody) orientovaným grafem, kde uzly odpovídají příkazům v kódu a hrany značí možné přechody na následující uzel, [32] nebo se uzly už skládají ze základních bloků instrukcí (takových, kde je následovník jednoznačný). [33] Kromě toho graf obsahuje také speciální vstupní a výstupní uzel bez instrukcí.

Pro uzly v grafu si chceme pamatovat určitá *fakta*, která budeme modifikovat při průchodu grafem podle instrukcí. Pokud obsahuje uzel více vstupních hran, musí být provedeno spojení faktů. Pro smysluplné spojování musí být množina všech možných hodnot fakt částečně uspořádána a spolu s operací spojení tvořit matematickou strukturu *svaz*. [34] Pro každou dvojici faktů musí v množině existovat jejich supremum a infimum, spojování také bude asociativní, komutativní a idempotentní. Pro taint analýzu popisují fakta bezpečnost aktuálních hodnot – tvoří tím úplné uspořádání a supremem (resp. infimem) dvojice faktů je jednoduše ta méně (resp. více) bezpečná hodnota. V praxi ale budou fakta obsahovat i další informace a také jejich spojování bude komplikovanější. Na rozdíl od formální abstraktní interpretace budeme při větvení ignorovat samotnou podmínku a do obou větví vstoupí stejná fakta.

Teoreticky lze nahradit volání metod jejich definicemi a vytvořit globální graf, analýza by se ale takto snadno opět mohla stát příliš náročnou na zdroje. Navíc počítáme s použitím nástroje už od iniciálních fází vývoje (oprava chyb už během implementace je mnohem snadnější) a globální pohled nemusí být vždy výhodou (můžeme chtít včas odhalit budoucí problémy, přestože aktuálně přímé riziko nepředstavují). Proto budeme provádět analýzu lokálně a globálně zohledníme pouze shrnutí výsledku lokální analýzy.

4 Nástroj FindBugs a možnosti rozšíření

FindBugs [12] je volně dostupný nástroj pro detekci chyb v Java programech pomocí statické analýzy kódu. Kromě vlastního grafického rozhraní a rozhraní pro příkazový řádek nabízí integraci s vývojovými prostředími (Eclipse, NetBeans, IntelliJ IDEA), nástroji pro automatizaci sestavování aplikace (Ant, Maven, Gradle) i jiným software (Jenkins, SonarQube). Dokáže detekovat širokou škálu chyb a potenciálních problémů – obvykle se jedná o relativně jednoduchá pravidla (např. chybějící příkaz `break` ve větvi výrazu `switch`) a síla nástroje je především v jejich množství, dokáže ale použít i složitější analýzu k detekci problémů, které nemusí být snadno zřetelné ani při manuální kontrole kódu zkušenějším vývojářem (např. detekce možných dereferencí hodnoty `null`). Mezi volně dostupným software se jedná jistě o nejlépe použitelný statický analyzátor, je-li cílem skutečná detekce chyb a ne pouze prohřešků proti formátování apod. [35] Obsahuje také kategorii bezpečnostních chyb a dovede odhalit některé případy SQL injekce, XSS nebo procházení adresářem.

Protože jako častý důvod, proč statické analyzátory stále nejsou při vývoji hojně využívány, bylo označeno velké množství falešných detekcí (planých poplachů), filosofií FindBugs je tento problém potlačit i za cenu toho, že některé skutečné problémy odhaleny nebudou. To je pravděpodobně dobrá strategie, protože rozsáhlý software bude pravděpodobně i tak obsahovat velké množství

chyb, které nástroj detekovat dokáže a použití analyzátoru může být velkým přínosem. Pro oblast bezpečnosti ale požadujeme vyšší garanci, že aplikace neobsahuje nebezpečné zranitelnosti, a není zde možné spoléhat na nástroj, který odhalí jen zlomek skutečně přítomných chyb.

FindBugs lze však rozšířit o další sady detektorů. Jednou z nich je i *FindSecurityBugs*, jejíž cílem je usnadnění bezpečnostního auditu kódu. [13] Tomu odpovídá i přítomnost detektorů, které nehledají přímo chyby, ale také některé samotné taint zdroje nebo místa, kde k bezpečnostním problémům často dochází. Projevilo se to i v detekci injekčních zranitelností – jako potenciální chyba byl původně značen každý taint sink, kromě případů, kdy vstupem zjevně byla konstanta známé hodnoty. Převažující falešná hlášení byla motivací pro implementaci výrazně pokročilejší analýzy. Další detekované chyby často souvisí se špatným použitím kryptografie nebo slabými algoritmy, přítomny jsou také detektory pro OS Android.

Analýza ve skutečnosti neprobíhá nad textovým zdrojovým kódem, ale FindBugs analyzuje rovnou přeložené třídy (`.class`) v podobě tzv. *bytekódu*, který už *JVM*[7] za běhu kompiluje do strojového kódu pro danou platformu. Pro implementaci rozšíření je tedy nutné se seznámit s instrukční sadou a principem fungování virtuálního stroje. [36] Kromě aplikací v Javě lze v omezené míře hledat chyby i v jazycích Scala, Groovy nebo Closure a dalších, pro něž existuje implementace pro JVM (např. Python a Ruby). Další výhodou je, že stejný bytekód odpovídá více variantám zápisu téhož programu a to navíc takové variantě, ze které je explicitně vidět, co program skutečně vykonává. Není tak nutné řešit formátování a různý zápis se stejným významem a tzv. syntaktický cukr. Dále máme jistotu, že se jedná o syntakticky validní program, a překladač také provede určitá zjednodušení (jako vyhodnocení konstantních výrazů), které mohou analýzu usnadnit.

4.1 Tvorba vlastních detektorů

Nejpřínosnějším nalezeným informačním zdrojem o tvorbě detektorů je prezentace [33] z konference *PLDI*[8] spolu se stručnou dokumentací [37] aplikačního rozhraní, pro dosažení potřebné úrovně znalostí však bylo často nutné procházet přímo zdrojové kódy nástroje FindBugs [38]. Vlastní detektor musí být přidán do rozšíření, což je JAR soubor obsahující nejméně jednu třídu s detektorem a dva konfigurační XML soubory obsahující informace o rozšíření, detektorech a samotných chybách.

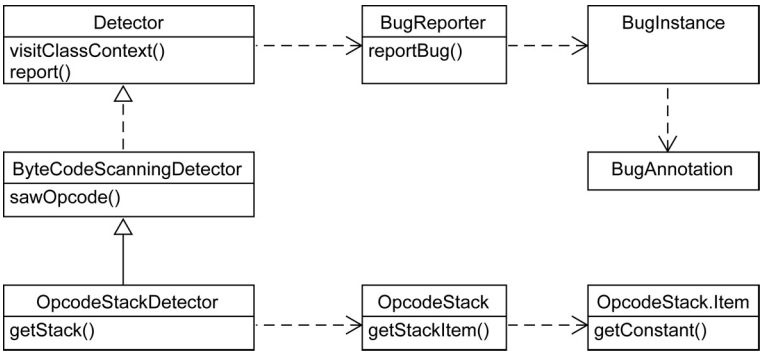
Samotný detektor je třída implementující rozhraní `Detector` (popř. `Detector2`) z balíku `edu.umd.cs.findbugs`. To předepisuje metodu `visitClass-`

[7] *Java Virtual Machine*, virtuální stroj jazyka Java

[8] *Programming Language Design and Implementation*, tradiční každoroční konference o návrhu a implementaci programovacích jazyků

`Context`, která je zavolána jedenkrát pro každou analyzovanou metodu. Je na daném detektoru, aby předanou instanci typu `ClassContext` zpracoval, např. si postupně vyžádal bytekód jednotlivých metod a vyhledal problematické posloupnosti instrukcí. Při nalezení chyby jsou použity instance tříd `BugReporter` a `BugInstance`, případně také `BugAnnotation` pro přidání dalších informací o chybě.

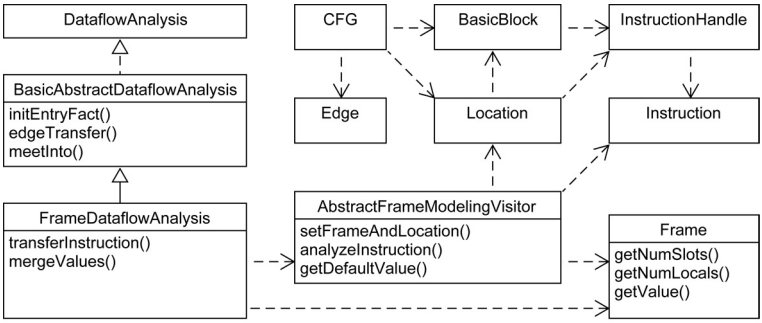
Přímou implementací rozhraní lze dosáhnout libovolné analýzy kódu, pro snadnější tvorbu detektorů ale existuje množství tříd s užitečnou funkcionalitou, které detektor může rozšířit (místo přímé implementace), popř. jen použít během analýzy. Jednodušší typy chyb lze obvykle detekovat jako posloupnost instrukcí bytekódu v rámci jedné metody. V tomto případě je vhodné, aby detektor rozšiřoval třídu `BytecodeScanningDetector`, která kromě množství užitečných funkcí obsahuje především metody volané při průchodu různými částmi kódu (např. třída, metoda, pole nebo přímo instrukce). Detektory tak lze programovat s využitím návrhového vzoru *návštěvník (visitor)*. Místo implementace metody `visitClassContext` může detektor přepsat přímo metodu `sawOpcode`, která bude zavolána pro každou analyzovanou instrukci spolu s číslem, které ji identifikuje. Pokud se jedná např. o instrukci typu `invoke`, přes další volání lze zjistit název metody nebo její signaturu (typy argumentů). Třída `OpcodeStackDetector` dále přidává metodu `getStack`, která se snaží modelovat zásobník operandů v místě volání (modelem platným nezávisle na skutečném běhu programu) a například umožňuje zjistit hodnoty konstantních parametrů metody. Závislosti zmíněných tříd jsou vyznačeny na obrázku 1.



Obr. 1: Zjednodušený diagram tříd pro detekci

4.2 Pokročilá analýza toku dat

Jednoduchou analýzu toku provádí už `OpcodeStackDetector`, když vyvozuje určitá fakta o hodnotách na zásobníku. Výhodnější je ale použít mechanismus, který zajistí vyšší přesnost, flexibilitu a znovupoužitelnost analýzy. FindBugs umožňuje předzpracovat kód metod a vytvořit *graf kontrolního toku (CFG)*, je-



Obr. 2: Zjednodušený diagram tříd pro analýzu toku

hož uzly jsou reprezentovány instancemi třídy `BasicBlock` a hrany objekty typu `Edge`. Konkrétní lokace v kódu reprezentují instance třídy `Location`, které se odkazují na svoje instance `BasicBlock` a `InstructionHandle` obsahující instrukci.

Pro každou metodu si lze vyžádat tok dat pro určitý typ analýzy, který může být využit ve více detektorech, prakticky se jedná o instanci potomka třídy `Dataflow`. Ve vytvářeném detektoru lze získat iterátor lokací pro danou metodu, během iterování zavolat na toku metodu `getFactAtLocation` (kde parametrem je právě lokace) a obdržet objekt, který nese fakta pro dané místo. O jaký fakt se jedná, záleží na typu toku, např. pro zjištění hodnot parametrů lze použít `ConstantDataflow`). Důležitá je možnost vytvoření vlastní analýzy toku dat a existence hierarchie tříd, která implementaci ulehčuje. Stačí pak implementovat metody jako `initEntryFact` pro inicializaci na začátku vstupu CFG, `edgeTransfer` pro případnou změnu faktu při přechodu přes hranu grafu nebo `meetInto`, která na začátku bloku kódu spojí fakta pocházející z různých vstupních hran do jednoho. Základní analýza mění fakta v uzlech jen na úrovni celého bloku, potomci `AbstractDataflowAnalysis` i na úrovni instrukcí v závislosti na implementaci metody `transferInstruction`.

Pokud chceme modelovat informace o jednotlivých hodnotách v programu, použitým faktem analýzy nemůže být abstrakce jedné hodnoty na lokaci, ale model lokálních proměnných a všech hodnot na zásobníku operandů. Pro tento účel je výhodné rozšířit abstraktní třídu `FrameDataflowAnalysis` a pro reprezentaci faktu využít potomka generické třídy `Frame`, kde teprve typový parametr reprezentuje fakt už pouze pro jednu hodnotu. Volání `transferInstruction` je vhodné volání delegovat na vlastního potomka třídy `AbstractFrameModelingVisitor`, který zajistí modifikaci faktů podle dané instrukce, opět s využitím vzoru návštěvník. Výchozí implementace už zajišťuje přesun faktů mezi proměnnými a zásobníkem a pro všechny instrukce modeluje aspoň odebírání a vkládání správného počtu položek (s výchozí hodnotou) na zásobník. Chování pro každou instrukci lze určit přepsáním odpovídající metody (např. `visitLDC` pro instrukci `ldc`). Závislosti zmíněných tříd lze vidět z obrázku 2. Analýza probíhá pouze lo-

kálně v rámci jedné metody, ale FindBugs umožňuje zvolit pořadí analyzovaných metod topologicky podle grafu vzájemného volání a s průběžnou sumarizací výsledků lze docílit iluze globální analýzy.

5 Implementace taint analýzy

Původní detektory injekcí ve FindSecurityBugs používaly existující `Constant-Dataflow`, který umožňuje zjistit konkrétní hodnotu položky na zásobníku. Pokud je známá, zjevně se o taint zdroj nejedná, analýza ale nezachytí případy, kdy sice hodnota známá není, přesto je zjevně konstantní (např. může nabývat několik různých hodnot uvedených v programu). Implementujeme proto vlastní `TaintDataflow`, kde fakt bude reprezentovat třída `Taint` s hlavními stavy *tainted* (nebezpečný ve smyslu taint analýzy), *unknown* (neznámý) a *safe* (bezpečný, opak *tainted*). V případě jedné z instrukcí `invoke` je vytvořen plný název volané metody (tj. včetně jména třídy, balíku a signatury) a je zjištěno, zda toto volání patří mezi taint zdroje resp. zdroje bezpečných dat. Pokud ano, je použit stav *tainted*, resp. *safe*. Bezpečná je hodnota také např. v případě instrukce `ldc`. V případě neznámého volání či instrukcí nepodstatných pro analýzu je použit výchozí stav *unknown*.

Spojování fakt probíhá vždy konzervativně – výsledný stav je vždy nejméně tak nebezpečný jako každá ze vstupních hodnot (např. spojením stavů *unknown* a *safe* bude opět *unknown*). Pokud bychom stavy uspořádali jako *tainted* > *unknown* > *safe*, pak výsledkem spojením stavů bude vždy jejich maximum. U mnohých volání bezpečnost návratové hodnoty závisí na jejích parametrech (včetně samotné instance – implicitního parametru každé nestatické metody). Např. metody `trim` či `toLower` třídy `String` přímo zachovávají stav na výstupu, metoda `concat` pak musí provést spojení stavů instance a parametru, to můžeme provést stejným způsobem jako při spojování větví v CFG.

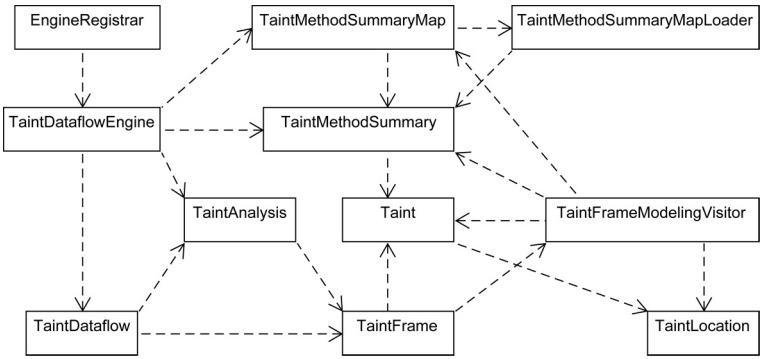
Analýzu řetězců usnadňuje fakt, že třída `String` je neměnitelná (tzv. *immutable*), velmi časté je ale např. spojování řetězců pomocí operátoru `+`. Pro takové případy je z důvodu efektivity vytvořena nová instance měnitelné třídy `StringBuilder`, na ní následně zavolány metody `append` pro každou část oddělenou znakem `+` a na závěr je vytvořen řetězec pomocí volání `toString`. Aby byla analýza funkční, musí být instance `Taint` vytvořená spojením informace o instanci třídy `StringBuilder` a parametru `append` nejen uložena na zásobník jako návratová hodnota, ale také musí být změněn stav samotné instance. S tímto se však pojí několik problémů – při změně položky na zásobníku např. musíme aktualizovat také odpovídající lokální proměnné nebo musí konstruktor správně inicializovat hodnoty na zásobníku. U neznámých metod také musíme předpokládat, že mohou modifikovat své měnitelné argumenty.

Aby se omezilo reportování volání s ošetřenými daty, specifické metody dále aktualizují instanci `Taint` přidáním informace o provedené konverzi. Jsou defino-

vány *tagy* (implementačně výčtový typ), které značí provedení konverze určitého znaku (např. `CR_ENCODED`) nebo skupiny (např. `XSS_SAFE`). U známých konverzních metod definujeme tagy, které daná metoda přidává či odebírá a instance `Taint` uchovává informaci, které tagy aktuální hodnota obsahuje. Kromě toho analýza podporuje také jednoduchou konverzi pomocí náhrady znaků (metody `replace` a `replaceAll` třídy `String`).

5.1 Konfigurace metod a automatické odvozování

Analýza spoléhá na množství informací o známých metodách, které jsou uloženy v mapování `TaintMethodSummaryMap`, kde klíčem je plný název metody (tzn. včetně třídy, balíku a signatury) a hodnotou instance třídy `TaintMethodSummary`. Třídy taint analýzy jsou zaneseny v obrázku 3. Kvůli tomu, že výstup metody může být závislý na vstupních parametrech, obsahuje třída `Taint` také kolekci indexů těchto parametrů. Pro případy, kdy nestačí výslednou instanci vložit na zásobník, ale přenést na objekt, který metodu volá, nebo jeho parametry, obsahuje `TaintMethodSummary` kolekci indexů do zásobníku, kde je nutné aktualizovat stav.



Obr. 3: Diagram tříd implementované taint analýzy

Instance `TaintMethodSummary` lze načítat z konfiguračních souborů, kde na každém řádku je plné jméno metody následované znakem `:` a konfiguračním řetězcem. Povinnou částí jsou indexy do zásobníku operandů, které určují položky pro sloučení k vytvoření nové instance `Taint`. Místo indexu je možné uvést do konfigurace také přímo stav (např. `SAFE`) a to i v kombinaci s indexy, pak bude výsledkem spojení všech těchto hodnot. Vzniklá instance `Taint` je ve výchozím stavu uložena na zásobník. Pokud ji dále chceme použít i na hodnoty hlouběji na zásobníku (na parametry či samotný objekt), jsou tyto indexy specifikovány za znakem `#`. Poslední částí konfiguračního řetězce je seznam tagů uvedených za znakem `|`. Názvu každého tagu předchází buď znak `+`,

nebo `-`, podle toho, jestli metoda tag přidává, nebo odebírá. Konfigurační soubory pro mnoho metod jsou obsaženy přímo uvnitř nástroje, předáním parametru `findsecbugs.taint.customconfigfile` lze dále zvenku konfiguraci přepsat nebo přidat vlastní metody bez nutnosti rekompilace nástroje.

Kvůli dědičnosti a polymorfismu nelze spolehlivě zjistit kód, který bude po zavolání metody skutečně vykonán, proto je ukládán název skutečně použité třídy do instance `Taint` při volání instrukce `new` a ta je přednostně použita při vyhledávání konfigurace metody. Konfigurace metod jsou také vyhledávány postupně pro všechny rodiče a implementovaná rozhraní, dokud není nalezena nějaká se známou konfigurací. Stačí proto konfigurovat pouze metodu stejného názvu (a signatury) jen u tříd či rozhraní, které jsou nejvýše v hierarchii dědičnosti. To podstatně zkrátí konfiguraci metod přítomných v hlubších hierarchiích (např. pro kolekce), umožní případně konfiguraci snadno změnit na všech místech současně a automaticky jsou pokryty implementace neznáme v době psaní konfigurace.

Veškerá taint analýza probíhá pouze lokálně v rámci jedné metody, tok dat ale v aplikacích často prochází skrz více metod i tříd. Proto analýza kromě využití předem nakonfigurovaného chování také odvozuje a ukládá instance `TaintMethodSummary` za běhu. Za tímto účelem si instance `Taint` se stavem *unknown* udržují kolekci indexů vstupních parametrů právě analyzované metody, na jejichž stavu je výsledná instance závislá. Na začátku analýzy je tato kolekce inicializována pouze indexem parametru pro odpovídající část proměnných a při každém spojení instancí `Taint` je výstupní množina dána sjednocením množin vstupních instancí.

Metoda s referenčním návratovým typem je běžně zakončena instrukcí `areturn`. Instanci `Taint`, která se v době jejího volání nachází na vrcholu zásobníku, lze přidat do `TaintMethodSummary` a uložit do mapování s původně jen ručně nakonfigurovanými metodami pod klíčem aktuálně analyzované metody. Protože metoda může vrátit hodnotu na více místech, dochází samozřejmě ke spojení všech možných návratových hodnot. Takto mohou být odvozeny nové taint zdroje či bezpečné zdroje (pokud hodnota nezávisí na argumentech), ale i metody, jejichž nebezpečnost závisí na vstupech, a metody provádějící konverzi.

5.2 Detektory využívající taint analýzu

Dosud zmíněné třídy analýzy pouze uloží potřebná fakta o všech lokacích v kódu, pro samotné reportování využíváme hierarchii detektorů. Nejvýše je `AbstractTaintDetector`, který přímo implementuje rozhraní `Detector`, pro každou metodu získá instanci `TaintDataflow` a pro každou lokaci CFG volá svoji abstraktní metodu `analyzeLocation` s odkazem na analyzovanou instrukci, instancí `TaintFrame` a dalšími parametry. Přímým potomkem je `AbstractInjectionDetector`, který rozšiřuje analýzu toku dat pro konkrétní typy chyb a provádí

Tabulka 1: Počty typů detekovaných volání pro injekční chyby

CWE	FSB 1.4.0	FSB 1.4.6	Přidáno
15	0	1	1
22	11	12	1
78	8	14	6
79	0	38	4
89	17	160	24
90	4	45	9
113	0	11	11
117	0	337	337
601	2	6	4
643	2	29	27
celkem	44	653	424

Tabulka zobrazuje, kolik volání známých jako taint sink byla analýza ve FindSecurityBugs schopná detekovat v původní a vylepšené verzi. Sloupec *Přidáno* značí počet přidaný v rámci této práce (zbytek rozdílu přidal původní autor rozšíření).

reportování chyb způsobem, který umožní zobrazit nejen zranitelný taint sink, ale pokud možno také taint zdroje a místa kódu, která značí cestu mezi taint zdroji a konečným voláním. `BasicInjectionDetector` pak umožňuje snadno vytvářet detektory pouze konfigurací potenciálně zranitelných argumentů metod. Kromě převodu původních detektorů z FindSecurityBugs byly přidány detekce nových chyb a podstatně zvýšen počet detekovaných typů volání (viz tabulka 1).

Taint analýza sice dokáže odvodit nové konfigurace, problém však nastane, pokud je taint sink volán s hodnotou, jejíž stav závisí na argumentu právě analyzované metody. Taint sink závislý na argumentu v tomto případě nebude ihned reportován, ale budou odchytávány volání metody, která taint sink obsahuje, a pro hodnoty nezávislé na dalších argumentech může být změněna priorita, se kterou bude chyba hlášena (a případně dále rekurzivně až po skutečný taint sink). Stejně jako při taint analýze je i v detektorech použito pořadí analýzy metod podle grafu volání. Protože volání metody nelze vždy spolehlivě detekovat, nebylo by vhodné chybu vůbec nereportovat, pokud není nalezeno, navíc může být žádoucí hledat i chyby v momentálně nevyužívaném kódu. Pokud je ale detekován nenulový počet volání s bezpečnou hodnotou a žádná s nebezpečnou či neznámou, předpokládáme, že volání detekovat dokážeme a o zranitelnost se pravděpodobně nejedná. V takovém případě je chyba reportována pouze s nízkou prioritou.

6 Vyhodnocení úspěšnosti detekce

Pro praktické ověření schopností analýzy vylepšených detektorů FindSecurityBugs a srovnání s jejich původní verzí a samostatným nástrojem FindBugs byla

zvolena sada *Juliet Test Suite*. Jedná se o testovací sadu agentury NSA[9] vyvinutou přímo pro vyhodnocení schopností nástrojů pro statickou analýzu kódu, dostupná je ze stránek laboratoře NIST[10]. [39] Sada se skládá z více než 25 tisíc testovacích případů (*test cases*) rozdělených do 112 skupin podle identifikátoru CWE, z toho 12 skupin s celkem necelými 8 tisíci případy odpovídá chybám typu injekce.

Každý případ se skládá z jedné či více tříd a kromě chyby pokud možno obsahuje také odpovídající kód bez zranitelnosti. Úspěšnost budeme posuzovat podle *citlivosti*, tedy toho, zda nástroj v daném případu problém odhalí (*true positive*), ale také podle tzv. *specificity* – zda chyba nebude reportována v místě, kde se ve skutečnosti nenachází (*true negative* – nereportovaná neexistující chyba). Z těchto metrik se dají odvozovat další jako *přesnost (accuracy)* udávající poměr počtu správných rozhodnutí o přítomnosti dané chyby oproti rozhodnutím nesprávným, *preciznost (precision)* značící relativní četnost skutečných chyb mezi detekcemi nebo F_1 *skóre* vypočítané jako harmonický průměr citlivosti a preciznosti, které může být použito jako ukazatel celkové úspěšnosti pouze pomocí jedné hodnoty. [40]

Výsledky pro samostatný FindBugs a FindSecurityBugs v různých verzích a nastaveních jsou pro jednotlivé typy chyb zobrazeny v tabulce 2 a průměrný

Tabulka 2: Citlivost a specificita detekce injekčních chyb

CWE-ID	Případů	FB	FSB 1.4.0	FSB 1.4.6 V	FSB 1.4.6 V+N
15	444	0 / 100	0 / 100	89 / 100	100 / 89
22*	888	5 / 100	100 / 16	89 / 100	100 / 89
78	444	0 / 100	100 / 0	89 / 100	100 / 89
79**	1 332	4 / 100	0 / 100	89 / 100	100 / 89
89	2 220	84 / 49	0 / 100	89 / 100	100 / 89
90	444	0 / 100	0 / 100	89 / 100	100 / 89
113	1 332	4 / 100	0 / 100	89 / 100	100 / 89
601	333	0 / 100	100 / 30	89 / 100	100 / 89
643	444	0 / 100	0 / 100	89 / 100	100 / 89

Tabulka zobrazuje citlivost a specificitu analýzy v procentech pro jednotlivé chyby typu injekce, které umožňuje sada Juliet testovat, a to pro samostatný FindBugs bez rozšíření (v nejnovější verzi 3.0.1), původní verzi FindSecurityBugs (před zapojením se do vývoje) a vylepšenou verzi FindSecurityBugs s různým nastavením hranice priority (pouze vysoká, nebo vysoká a normální, detekce s nízkou prioritou jsou ignorovány). Hodnota *0 / 100* např. značí, že nebyla hlášena žádná chyba, naopak *100 / 0* značí detekci všech skutečných chyb, ale také považování všech bezpečných volání za chybu.
*Výsledek pro CWE-22 byl získán sjednocením skupin pro CWE-23 a CWE-36
**CWE-79 je dáno sjednocením CWE-80, CWE-81 a CWE-83
Pozn.: CWE-113 je kvůli nižší nebezpečnosti celkově reportováno s nižší prioritou, pro zjednodušení však budeme předpokládat stejný systém jako pro ostatní chyby.

[9] *National Security Agency*, Národní bezpečnostní agentura, vládní organizace USA
[10] *National Institute of Standards and Technology*, Národní institut standardů a technologie

Tabulka 3: Srovnání celkové úspěšnosti detekce injekcí

	FB	FSB 1.4.0	FSB 1.4.6 V	FSB 1.4.6 V+N
Citlivost	11	33	89	100
Specificita	94	72	100	89
Přesnost	53	53	95	95
Preciznost	65	54	100	90
F_1 skóre	19	41	94	95

Zde je zobrazena celková úspěšnost detekce pro FindBugs a různé verze FindSecurityBugs vypočtená z (neváženého) průměru citlivosti a specificity uvedených v tabulce 2, zbývající metriky jsou odvozeny z nich.

výsledek i s dalšími metrikami pak v tabulce 3. Častou překážkou při nasazování automatické statické analýzy je vysoká míra falešných poplachů a FindBugs se proto soustředí na vysokou specificitu (hlášené chyby by měly být skutečné). Výjimkou je zjevně detektor pro SQL injekce (CWE-89), který za cenu nižší specificity odhalil většinu chyb. Důvodem je pravděpodobně častý výskyt a vážné důsledky zranitelností tohoto typu. Pokud připočítáme i detekce s nižší prioritou (nejsou zobrazeny v tabulce), citlivost se ještě zvýší, ale přesnost i preciznost klesnou na 50 %. Naopak FindSecurityBugs je zamýšlen jako nástroj pro pomoc při bezpečnostním auditu kódu a cílí na vysokou citlivost i za cenu nižší specificity. Bohužel detekce množství injekčních zranitelností byla implementována pouze pro případy, které se v testovací sadě nenachází (jiná rozhraní pro SQL, LDAP a XPath injekce, XSS v JSP), popř. vůbec. Celková přesnost detekce je shodou náhod pro FindBugs i původní FindSecurityBugs téměř stejná (53 %), F_1 skóre je však pro FindBugs výrazně nižší kvůli nízké citlivosti (19 % oproti 41 %).

Nová verze FindSecurityBugs (1.4.6) přistupuje k analýze všech chyb typu injekce jednotně (i když specifika chyb jako XSS vyžadovala přizpůsobení) a úspěšnost detekce jednotlivých typů chyb v sadě je shodná. Cílem bylo vytvořit analýzu s ideálně absolutní citlivostí, přitom ale podstatně omezit množství falešných poplachů a pro ještě lepší praktickou použitelnost také potvrdit vybrané detekce, u nichž je vysoká pravděpodobnost zneužitelné chyby. Ty chceme reportovat s vyšší prioritou pro možnost použití nástroje jako analyzátoru s vysokou precizností. Z výsledků vidíme, že všechny detekce s vysokou prioritou jsou chybami skutečnými (100% specificita a preciznost) a to při zachování vysoké citlivosti 89 %. Po přidání detekcí s normální prioritou je citlivost absolutní a přesto 90 % hlášení odpovídá skutečným chybám. Přesnost analýzy byla výrazně zvýšena na 95 % pro oba způsoby nastavení. Podobné jsou také hodnoty F_1 skóre, to vychází mírně vyšší pro nastavení zahrnující také problémy s normální prioritou (95 % oproti 94 %).

6.1 Rozbor výsledků

Pro hlubší porozumění výsledkům je však vhodné se seznámit se strukturou testovací sady. Testovací případy nebyly vytvořeny ručně, ale vygenerovány za použití šablon kombinujících různé varianty toku a funkční varianty chyby. V případě injekčních zranitelností funkční variantu určuje konkrétní taint sink a taint zdroj a testový případ existuje pro každou kombinaci, takže jejich počet je relativně vysoký. Pro úspěšnou analýzu případu proto musí nástroj rozpoznat taint zdroj (popř. bezpečný zdroj) i taint sink a porozumět použité variantě toku. Při testu falešných detekcí je jako zdroj použit konstantní řetězec, nebo je daný taint zdroj ošetřen konverzí. Množství chyb nebylo odhaleno už proto, že daný taint sink nebyl analýze známý (popř. neexistoval vůbec detektor pro odpovídající chybu). Tam, kde je citlivost v tabulce 2 nenulová, byl ale každý testovaný taint sink podporován.

Pro FindBugs je hlavním důvodem nízké citlivosti to, že kromě SQL injekce považuje za taint zdroj pouze metodu `getParameter` (třídy `HttpServletRequest`) a ignoruje dalších 11 testovaných zdrojů. Naopak FindSecurityBugs taint zdroje nerozlišoval a za bezpečné považoval pouze snadno detekovatelné konstanty, takže nenulovou specificitu má pouze v případech, kdy taint sink přímo přejímá hodnotu taint zdroje (resp. konstanty). Nová verze FindSecurityBugs zná všechny testované taint zdroje (a množství netestovaných) pro detekci s vysokou prioritou a obsahuje pokročilou detekci konstantních i ošetřených vstupů.

Obtížnost analýzy toku někdy ovlivňuje už samotná funkční varianta – např. chyba typu procházení adresářem se v sadě nachází jako *relativní* (CWE-23) i *absolutní* (CWE-36), při první variantě ale taint sink obsahuje zřetězení zdroje a konstanty (navíc závislé na vyhodnocení podmínky pro vyhodnocení operačního systému), takže je detekce složitější. V testu některých chyb jsou bezpečné zdroje dat tvořeny konverzí, což je těžší případ než pouhé volání s konstantou. Kromě toho je ale každý testový případ ovlivněn jednou ze 37 variant toku.

Všechny analyzátory si poradí s prvními 17 variantami obsahující různé větvení nebo kopií dat v rámci metody (varianta 31), v ostatních případech je ale nutné sledovat tok skrz více metod, popř. i tříd. Zde jsou např. taint zdroje zapouzdřeny v jiné metodě nebo je taint sink v metodě, která je teprve volána s nebezpečným argumentem (a to až přes 4 třídy ve variantě 54). Data nejsou vždy poslána přímo, ale někdy nejprve uložena do pole či kolekce, popř. přetypována. Obsaženo je také volání abstraktní metody, kde zneužitelnost chyby závisí na volbě instance konkrétní třídy (varianta 81). Jedinou variantu, kterou v některých detektorech dokáže analyzovat FindBugs a nová verze FindSecurityBugs zatím ne, je předání dat přes privátní atribut třídy. Tři varianty nedokázal správně analyzovat žádný detektor – data poslaná uvnitř instance vlastní třídy nebo v serializované podobě a předání přes veřejný atribut. Zbývajících 33 vari-

ant nová verze FindSecurityBugs správně rozpozná, pokud ignorujeme detekce reportované s nízkou prioritou (s nimi by byly hlášeny chyby i v bezpečných voláních pro 15 variant toku). Sada Juliet samozřejmě netestuje schopnost analýzy toku vyčerpávajícím způsobem a množství technik přidaných nově do FindSecurityBugs nevedlo automaticky k vyšší přesnosti v testu, pro analýzu reálného kódu je významná např. podpora velkého množství rozhraní a automatické odvozování přenosu nebezpečných dat nejen pro taint zdroje, ale i další metody.

7 Závěr

Tato práce řešila problematiku využití automatické statické analýzy pro detekci injekčních zranitelností. Představili jsme známe typy injekcí, vysvětlili princip taint analýzy a také popsali možnosti rozšíření nástroje FindBugs. Implementovaná taint analýza modeluje hodnoty dat v zásobníku operandů a lokálních proměnných pro každé místo v kódu, při čemž naráz uvažuje všechny možnosti větvení. Pro každou položku si pamatuje především to, zda data v ní mohou být nebezpečná (alespoň při určitém větvení) a zda byla ošetřena konverzí. Jednotlivé instrukce tyto příznaky mění a přenášejí mezi sebou, takové chování je definováno i pro instrukce typu *invoke* a více než 750 nakonfigurovaných metod (navíc automaticky převzato potomky odpovídajících tříd či rozhraní). Konfigurace jsou také automaticky odvozovány pro neznámé metody na základě výsledku jejich analýzy, takže chyby mohou být hledány napříč celým programem, tomu napomáhá vhodná volba pořadí analýzy metod. Podrobnější informace o chybách, způsobu analýzy i další jsou obsaženy v diplomové práci [41].

Rozšíření FindSecurityBugs bylo také obohaceno o detekci tří dalších typů injekčních zranitelností a opraveno několik chyb. Existující detektory injekcí byly přepsány a provedena byla revize potenciálně zranitelných volání – pro každou chybu jsme nově přidali nejméně jeden taint sink a i s novými detektory byl jejich počet navýšen o více než 400. Všechny změny (podle systému GitHub asi 13 tisíc přidaných řádků) byly přijaty do oficiální distribuce FindSecurityBugs [42] a novější verzi nástroje jsme spolu s původním autorem ke konci roku 2015 představili na konferenci *Black Hat Europe* [43], do vylepšování vyvinutého mechanismu se už také zapojují další přispěvatelé.

Úspěšnost analýzy byla vyhodnocena na části sady Juliet – otestována byla detekce 12 typů injekčních chyb v necelých 8 tisících testovacích případech. Analýzu vylepšené verze FindSecurityBugs limitovaly pouze 4 náročné varianty toku (z 37), které byly rozeznány až při nižší prioritě (avšak za cenu reportování také bezpečných volání v těchto variantách). Nástroj je proto vhodné použít tak, že nejprve budou ověřeny chyby s vysokou prioritou, kde je nízká míra falešných detekcí (pro sadu Juliet nulová). Následně můžeme zkontrolovat detekce s nižší prioritou, které by měly obsahovat všechny skutečné chyby daného typu, ale

pravděpodobně i bezpečná volání (ovšem v mnohem menší míře než u původního rozšíření). Celková přesnost analýzy vzrostla na 95 % (pro obě nastavení hranice priority) oproti 53 % u původní verze rozšíření i samostatného nástroje FindBugs.

Literatura

[1] *CWE-120: Buffer Copy without Checking Size of Input (‚Classic Buffer Overflow‘)* 2014. https://cwe.mitre.org/data/definitions/120.html (cit. 08. 05. 2016)

[2] *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component (‚Injection‘)*. 2015. https://cwe.mitre.org/data/definitions/79.html (cit. 08. 05. 2016)

[3] *CWE-89: Improper Neutralization of Special Elements used in an SQL Command (‚SQL Injection‘)*. 2015. https://cwe.mitre.org/data/definitions/89.html (cit. 08. 05. 2016)

[4] *CWE-90: Improper Neutralization of Special Elements used in an LDAP Query (‚LDAP Injection‘)*. 2015. https://cwe.mitre.org/data/definitions/90.html (cit. 08. 05. 2016)

[5] *CWE-643: Improper Neutralization of Data within XPath Expressions (‚XPath Injection‘)*. 2014. https://cwe.mitre.org/data/definitions/643.html (cit. 08. 05. 2016)

[6] *CWE-93: Improper Neutralization of CRLF Sequences (‚CRLF Injection‘)*. 2015. https://cwe.mitre.org/data/definitions/93.html (cit. 08. 05. 2016)

[7] *CWE-77: Improper Neutralization of Special Elements used in a Command (‚Command Injection‘)*. 2015. https://cwe.mitre.org/data/definitions/77.html, (cit. 08. 05. 2016)

[8] *CWE-79: Improper Neutralization of Input During Web Page Generation (‚Cross-site Scripting‘)*. 2015. https://cwe.mitre.org/data/definitions/79.html (cit. 08. 05. 2016)

[9] *CWE-22: Improper Limitation of a Pathname to a Restricted Directory (‚Path Traversal‘)*. 2015. https://cwe.mitre.org/data/definitions/22.html (cit. 08. 05. 2016)

[10] *CWE-601: URL Redirection to Untrusted Site (‚Open Redirect‘)*. 2015. https://cwe.mitre.org/data/definitions/601.html (cit. 08. 05. 2016)

[11] *Static Code Analysis*. 2016.
https://www.owasp.org/index.php/Static_Code_Analysis (cit. 08. 05. 2016)

[12] *FindBugs – Find Bugs in Java Programs*. 2015.
http://findbugs.sourceforge.net/ (cit. 08. 05. 2016)

[13] *Find Security Bugs*. 2015. https://find-sec-bugs.github.io/ (cit. 08. 05. 2016)

[14] *CWE – Common Weakness Enumeration*. 2015.
https://cwe.mitre.org/index.html (cit. 08. 05. 2016)

[15] Christey, S.: *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. https://cwe.mitre.org/top25/index.html (cit. 08. 05. 2016)

[16] *Top 10 2013*. 2016.
https://www.owasp.org/index.php/Top_10_2013 (cit. 08. 05. 2016)

[17] *CAPEC-7: Blind SQL Injection*. 2015.
https://capec.mitre.org/data/definitions/7.html (cit. 08. 05. 2016)

[18] Damele, B., Stampar, M.: *sqlmap: Automatic SQL injection and database takeover tool*. 2016. http://sqlmap.org/ (cit. 08. 05. 2016)

[19] *CWE-117: Improper Output Neutralization for Logs*. 2014.
https://cwe.mitre.org/data/definitions/117.html (cit. 08. 05. 2016)

[20] *CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers („HTTP Response Splitting“)*. 2015.
https://cwe.mitre.org/data/definitions/113.html (cit. 08. 05. 2016)

[21] *Top 10 2013-A10-Unvalidated Redirects and Forwards*. 2013.
https://www.owasp.org/index.php/
Top_10_2013-A10-Unvalidated_Redirects_and_Forwards (cit. 08. 05. 2016)

[22] *Category: OWASP Enterprise Security API*. 2015.
https://www.owasp.org/index.php/ESAPI (cit. 08. 05. 2016)

[23] *OWASP Java Encoder Project*. 2015.
https://www.owasp.org/index.php/OWASP_Java_Encoder_Project
(cit. 08. 05. 2016)

[24] *Class StringEscapeUtils*. 2015.
https://commons.apache.org/proper/commons-lang/apidocs/org/
apache/commons/lang3/StringEscapeUtils.html (cit. 08. 05. 2016)

[25] *Class HtmlUtils*. 2016.
http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/util/HtmlUtils.html (cit. 08. 05. 2016)

[26] *OWASP Testing Guide Appendix C: Fuzz Vectors*. 2014.
https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors (cit. 08. 05. 2016)

[27] *perlsec – Perl Security*. 2016.
http://perldoc.perl.org/perlsec.pdf (cit. 08. 05. 2016)

[28] Cadar, C., Sen, K,: *Symbolic Execution for Software Testing: Three Decades Later*. 2013. http://srl.cs.berkeley.edu/∼ksen/papers/cacm13.pdf (cit. 08. 05. 2016)

[29] Filliâtre, J.-C.: *Deductive software verification*. 2011.
http://link.springer.com/article/10.1007/s10009-011-0211-0 (cit. 08. 05. 2016)

[30] Jhala, R., Majumdar, R.: *Software Model Checking*. 2009.
http://goto.ucsd.edu/∼rjhala/papers/software_model_checking_survey.pdf (cit. 08. 05. 2016)

[31] Schwartzbach, M. I.: *Lecture Notes on Static Analysis*.
http://lara.epfl.ch/w/_media/sav08:schwartzbach.pdf (cit. 08. 05. 2016)

[32] Schwartzbach, M. I.: *CS252r Spring 2011, Data Flow Analysis*. 2011.
http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf (cit. 08. 05. 2016)

[33] *Using FindBugs for Research*. 2007.
https://findbugs-tutorials.googlecode.com/files/uffr-talk.pdf (cit. 08. 05. 2016)

[34] Insall, M., Weisstein, E. W.: *Lattice, From MathWorld – A Wolfram Web Resource*. http://mathworld.wolfram.com/Lattice.html (cit. 08. 05. 2016)

[35] Formánek, D.: *Nástroje pro statickou a dynamickou analýzu kódu se zaměřením na bezpečnostní chyby*. bakalářská práce, Masarykova univerzita, 2014.

[36] Lindholm, T. et al.: *The Java Virtual Machine Specification, Java SE 8 Edition*. 2015. http://docs.oracle.com/javase/specs/jvms/se8/html/index.html (cit. 08. 05. 2016)

[37] *FindBugs API Documentation*. 2015.
http://findbugs.sourceforge.net/api/index.html (cit. 08. 05. 2016)

[38] *The new home of the FindBugs project*. 2016.
https://github.com/findbugsproject/findbugs (cit. 08. 05. 2016)

[39] *Test Suites*. 2015.
https://samate.nist.gov/SRD/testsuite.php (cit. 08. 05. 2016)

[40] François, D.: *Binary classification performances measure cheat sheet*. 2009.
http://www.damienfrancois.be/blog/files/modelperfcheatsheet.pdf
(cit. 08. 05. 2016)

[41] Formánek, D.: *Detekce bezpečnostních chyb pomocí statické analýzy kódu*. Masarykova univerzita, diplomová práce, 2016.

[42] *Find Security Bugs*. 2016. https://github.com/find-sec-bugs/find-sec-bugs
(cit. 08. 05. 2016)

[43] *Black Hat Europe 2015, Arsenal*. 2015.
https://www.blackhat.com/eu-15/arsenal.html#findsecuritybugs
(cit. 08. 05. 2016)

Optimalizovaná batéria štatistických testov NIST STS

M. Sýs, Z. Říha, V. Matyáš

E-mail: syso@fi.muni.cz

Abstrakt

Nielen v kryptografii ale v množstve ďalších odvetví sa vyžaduje generovanie náhodných a nepredikovateľných čísel. Medzi najdôležitejšie nástroje na analýzu náhodnosti dát/generátorov patrí batéria štatistických testov NIST STS. V praxi sa posudzuje kvalita generátora testovaním veľkého množstva dát, ktoré daný generátor vygeneruje, čo kladie vysoké požiadavky na efektivitu implementovaných testov. Pôvodná implementácia NIST STS je neefektívna, čo znamená, že pre bežný objem dát (jednotky gigabajtov) trvá otestovanie generátora rádovo dni. V článku predstavíme novú implementáciu NIST STS, ktorá je 50× *rýchlejšia a dokáže otestovať 1 GB za 20 minút. Optimalizovaná batéria obsahuje novú verziu Berlekamp-Massey algoritmu pre konštrukciu minimálneho spätnoväzobného registra, ktorá je až* 187× *rýchlejšia ako pôvodná. V článku sa taktiež venujeme kľúčovej časti testovania a to interpretácii výsledkov. S využitím optimalizovanej verzie sme otestovali dáta z kvantového generátora a zistili sme, že generátor možno považovať za náhodný, ak neprejde 7 alebo menej testami z NIST STS.*

1 Úvod

Štatistické testy náhodnosti majú v kryptografii svoje nezastupiteľné miesto. Používajú sa nie len na testovanie náhodnosti dát, ale aj na testovanie dizajnu kryptografických funkcií. Štatistické testy náhodnosti sa používajú na certifikovanie deterministických pseudonáhodných a nedeterministických náhodných generátorov čísel. Pri testovaní dizajnu kryptografických funkcií (prúdové šifry, blokové šifry, hašovacie funkcie) sa testuje ich výstup, ktorý by mal byť štatisticky neodlíšiteľný od výstupu náhodných generátorov. Ďalšie nemenej významné využitie štatistických testov je pri power-up testoch, pri ktorých sa kontroluje funkčnosť opätovne spustených kryptografických moduloch.

Každý štatistický test analyzuje dáta zo špecifickej stránky – pomer frekvencie bitov 0 verzus 1, dĺžka najdlhšieho bloku zloženého z rovnakých bitov a pod. Štatistické testy náhodnosti testujú štatistickú zhodu s očakávanými výsledkami pre ideálny náhodný generátor. V praxi sa často stáva, že hoci generátor produkuje sekvencie s očakávaným pomerom 50 % vs 50 % nulových a jednotkových bitov (napr. 00000001111111), iný test odhalí neštandardný výskyt „vzorov“ iného typu (veľa 2 bitových blokov 00). Z tohoto dôvodu sa testy zgrupujú do komplexnejších „batérií“ testov, ktorá zväčša obsahuje aj sadu pseudonáhodných generátorov čísel. V súčasnosti medzi kryptograficky významné a často používané batérie patria: Diehard (novšia verzia Dieharder) [9], TestU01 [7], NIST STS [1]. Vo všeobecnosti pozostávajú tieto batérie z rôznych testov, hoci niektoré testy (Runs, Monobit) sú implementované vo viacerých batériách. Dieharder je batéria 27 testov prevažne používaná na testovanie fyzikálnych generátorov. TestU01 je v súčasnosti asi najkomplexnejší nástroj na testovanie náhodnosti. Vzhľadom na rôznu náročnosť testov TestU01 obsahuje 3 batérie testov: rýchly Small Crush (10 testov), stredne rýchly Crush (96 testov) a pomalý Big Crush (106 testov). Asi najpopulárnejšia je batéria NIST STS, nakoľko sa používa na testovanie binárnych postupností a ich certifikovanie. NIST STS pozostáva z 15 testov, ktoré analyzujú dáta z hľadiska bitov, blokov bitov (veľkosti < 30) a veľkých častí (veľkosti $> 1\,000$ bitov). Jej 3 testy (Monobit, Runs, Longest Run) boli spolu s Poker testom z Diehard batérie donedávna používané ako power-up testy [8].

2 Štatistické testy náhodnosti

Testy vychádzajú zo štatistického testovania hypotéz a testujú hypotézu generátor/vygenerované dáta sú náhodné. Výsledkom testu nie je exaktná odpoveď – testovaný generátor je zlý/dobrý, ale pravdepodobnostná – testovaný generátor je na 99 % dobrý. Dôvodom je fakt je, že aj skutočne náhodný generátor občas vyprodukuje sekvenciu bitov, ktorá sa zdá byť nenáhodná (napr. samé nuly). Čím „nenáhodnejšia“ (extrémnejšia) sa postupnosť zdá byť, tým je však pravdepodobnosť, že ju dobrý generátor vyprodukuje menšia.

Štatistické testy náhodnosti v podstate vyhodnocujú extrémnosť danej postupnosti tj. testy počítajú pravdepodobnosť (takzvanú p-hodnotu), s akou by skutočne náhodný generátor vyprodukoval danú resp. extrémnejšiu (menej náhodnú) postupnosť (vzhľadom na dané kritérium). Výpočet a význam p-hodnoty možno ilustrovať na nasledujúcom príklade.

Príklad 1 *Predpokladajme, že generátor vygeneroval postupnosť 30 bitov. Pre ideálny náhodný generátor (50 % vs 50 % pre generovanie bitu 0 resp. 1) je možné vypočítať pravdepodobnosť, že tento generátor vygeneruje konkrétny počet jedno-*

tiek resp. núl. Nasledujúca tabuľka zobrazuje percentuálne pravdepodobnosti, že dobrý náhodný generátor vygeneruje presne i jednotiek a $30 - i$ núl.

počet jednotiek	5	6	7	8	9	10	11	12	13	14	15
pravdepodobnosť	1	4	7	12	16	18	16	12	7	4	1

Ak generátor vygeneruje 13 jednotiek, potom p-hodnotu testu možno spočítať ako celkovú pravdepodobnosť extrémnejších výsledkov. Pod extrémnejším výsledkom možno intuitívne chápať počet jednotiek, ktorý je od ideálnej hodnoty 15 vzdialený viac. Teda p-hodnotu vypočítať ako sumu pravdepodobností pre (7 a menej) a (13 a viac) jednotiek. Teda p-hodnota by bola v našom prípade 0,26 *(26 %).*

Ak test vypočíta napr. p-hodnota= 0,01znamená to, že dobrý generátor takúto alebo ešte extrémnejšiu (menej „náhodnú“) postupnosť vygeneruje s pravdepodobnosťou 1 %. Posúdiť či daný výsledok pochádza z ešte z dobrého náhodného generátora (a mali sme jednoducho smolu, že vygeneroval tak extrémnu postupnosť), alebo už vychýleného generátora závisí na nás a na miere našej „tolerancie“. Pri testovaní sa postupuje tak, že najskôr sa stanoví táto „miera tolerancie“ (hladina významnosti α) a na jej základe sa zamieta resp. nezamieta hypotéza o dobrom generátore.

Celkovo možno testovanie náhodnosti rozdeliť do nasledovných krokov:

- voľba hladiny významnosti α – zväčša sa stanovuje na 1 %, 0,5 % resp. 0,1 %,
- získanie histogramu skúmaných vzorov – počítajú sa výskyty (frekvencie) jednotlivých vzorov (napr. histogram počtu dvojbitových blokov, $f_{00} = 10$, $f_{01} = 30$, $f_{10} = 20$, $f_{11} = 40$) v danej postupnosti,
- výpočet štatistiky – štatistika je číslo, ktoré reprezentuje mieru extrémnosti postupnosti vzhľadom na vypočítaný histogram ($S_{obs} = \sum |f_{ij} - 25| = 40$). Čím väčšia je hodnota štatistiky S_{obs}, tým extrémnejší (menej pravdepodobný) je histogram frekvencií a analyzovaná postupnosť. Pre $S_{obs} = 0$ je postupnosť a histogram v zhode s očakávanou ideálnou náhodnou postupnosťou.
- výpočet p-hodnoty – pravdepodobnosť, že dobrý generátor vygeneruje postupnosť s $S > S_{obs}$,
- Rozhodnutie o zamietnutí/nezamietnutí hypotézy – ak je p-hodnota menšia ako hladina významnosti α hypotézu zamietame a generátor považujeme za zlý/vychýlený. V opačnom prípade hypotézu nezamietame a generátor považujeme za náhodný vzhľadom na skúmané vzory.

Pri uvedenom postupe sa môžeme dopustiť 2 druhov chýb:

- chyba prvého druhu – dobrý generátor považujeme za zlý,
- chyba druhého druhu – zlý generátor považujeme za dobrý.

Pravdepodobnosť chyby prvého druhu je rovná voliteľnej hodnote α a je teda kontrolovateľná. Na druhej strane nevieme stanoviť hodnotu pravdepodobnosti chyby druhého druhu β. Vieme však, že tieto sú vzájomne prepojené: s klesajúcou pravdepodobnosťou chyby prvého druhu rastie pravdepodobnosť chyby druhého druhu a opačne (pri fixnej dĺžke postupnosti). Teda je vhodné stanoviť rozumnú hodnotu α, ktorá sa pri testoch náhodnosti bežne volí ako 1 %, 0,5 %, 0,1 %. Naraz možno zmenšiť pravdepodobnosti (α, β) oboch chýb tak, že zväčšíme veľkosť testovanej postupnosti. Preto sa pri testovaní generátorov náhodných čísel testujú spravidla gigabajty dát.

3 NIST STS

Batéria NIST STS pozostáva z 15 testov, ktoré analyzujú náhodnosť binárnych postupností z hľadiska bitov a blokov bitov. Všetky testy sú parametrizované bitovou veľkosťou vstupnej postupnosti n. Niektoré testy, ktoré analyzujú bloky bitov sú parametrizované ešte veľkosťou bloku parametrom m (malé bloky $m < 30$), M (veľké bloky $M > 500$). Testy možno rozdeliť do troch skupín:

- testy počítajúce štatistiku jednotlivých bitov – Frequency(Monobit), Frequency within a Block, Runs, Longest run of ones, Cumulative sums, Random Excursions, Random Excursions Variant,
- testy počítajúce štatistiku malých blokov bitov – Non-overlapping temaplate matching, Overlapping temaplate matching, Maurer‘ s Universal, Serial, Approximate Entropy,
- testy počítajúce komplexnú štatistiku veľkých blokov bitov – Rank, Spectral, Linear complexity.

Pri vyhodnocovaní testov sa používajú aproximácie, ktoré sú dostatočne presné až od určitej dĺžky postupnosti n, ktorá tiež môže závisieť na veľkosti analyzovaných blokov. Nasledujúca tabuľka zobrazuje odporúčané hodnoty n a m, M. Niektoré testy spúšťajú viaceré „podtesty“ Serial,Cumulative sums – 2, Random Excursions – 8, Random Excursions Variant – 18, Non-overlapping template matching – 148* (alebo menej – závisí na veľkosti bloku) a teda celkový počet spúšťaných testov môže byť až 188.

3.1 Optimalizácia NIST STS

Pri implementácii testov batérie NIST STS sa zrejme kládol dôraz na univerzálnosť implementácie (big endian, little endian) a nie na efektívnosť implementácie. Základný problém batérie je neefektívna reprezentácia dát, kde každý bajt ukladá jediný bit postupnosti. V našej práci [2, 3] sme sa zamerali na optimalizovanie batérie NIST STS. V minulosti už boli pokusy zefektívniť túto batériu [6], s výsledným zrýchlením 13× s výnimkou Spectral testu. My sme podobne ako [6]

Tabulka 1: Názvy testov a odporúčané hodnoty ich parametrov n, m a M

Test #	Názov testu	n	m resp. M
1.	Frequency	$n \geq 100$	
2.	Frequency within a Block	$n \geq 100$	$20 \leq M \leq n/100$
3.	Runs	$n \geq 100$	-
4.	Longest run of ones	$n \geq 128$	
5.	Rank	$n > 38\,912$	–
6.	Spectral	$n \geq 1\,000$	
7.	Non-overlapping T. M.	$n \geq 8m - 8$	$2 \leq m \leq 21$
8.	Overlapping T. M.	$n \geq 10^6$	
9.	Maurer's Universal	$n > 387\,840$	
10.	Linear complexity	$n \geq 10^6$	$500 \leq M \leq 5\,000$
11.	Serial		$2 < m < \lfloor \log_2 n \rfloor - 2$
12.	Approximate Entropy		$m < \lfloor \log_2 n \rfloor - 5$
13.	Cumulative sums	$n \geq 100$	
14.	Random Excursions	$n \geq 10^6$	
15.	Random Excursions Variant	$n \geq 10^6$	

reprezentovali dáta štandardným spôsobom, čo nám umožnilo pracovať s bitmi paralelne pomocou bitových operácií nad 32 resp. 64 bitovými slovami. Techniky použité pri optimalizácii možno popísať nasledovne:

- testy počítajúce štatistiku jednotlivých bitov – tieto testy boli optimalizované pomocou vyhľadávacích (lookup) tabuliek,
- testy počítajúce štatistiku malých blokov bitov – bola implementovaná rýchla (spracovanie 100 MB za 1 s) funkcia, ktorá vypočíta histogram m ($m < 25$) bitových blokov,
- testy počítajúce komplexnú štatistiku veľkých blokov bitov – testy Linear Complexity a Rank sme reimplementovali tak, že nová implementácia využíva celú šírku slova 32 resp. 64 a dokáže tak spracovávať bity „paralelne". V oblasti kryptografie známy Berlekamp-Massey algoritmus sme tak urýchlili až 187×. Optimalizáciou Berlekamp-Massey algoritmu sme podstatným spôsobom optimalizovali najpomalší test batérie Linear Complexity test, ktorý bol v práci [6] optimalizovaný len minimálne. Pre optimalizáciu Spectral testu sme využili rýchlu knižnicu FFTW počítajúcu Fourierovu transformáciu.

Viac podrobností o optimalizovanej verzii NIST STS možno nájsť v článku [2] a článku [3]. Tabuľka 2 ukazuje časy potrebné na spracovanie 20 MB dát pre originálnu a našu optimalizovanú verziu NIST STS. Novou batériou teda možno skrátiť testovanie generátorov 50× a teda 1 GB možno otestovať za cca 20 minút (namiesto 17 hodín). Implementáciu možno nájsť na stránke [4].

Tabulka 2: Časy (v milisekundách) behu testov originálnej NIST STS 2.1.2 a optimalizovanej verzie O-NIST STS a zrýchlenie O-NIST vs NIST STS 2.1.2

Test	m, M	NIST 2.1.2 (ms)	O-NIST (ms)	O-NIST zrýchlenie
Frequency (Monobit)		137	6	22.3
Frequency Within a Block	128	58	11	5.2
Runs		1 183	14	86.4
Longest Run of Ones in a Block		651	28	23.1
Binary Matrix Rank		3 588	307	11.7
Spectral		18 700	18 738	1.0
Non-overlapping Template	9	125 862	107	1178.7
Overlapping Template	9	1 355	61	22.1
Maurer‘s Universal		2 758	154	17.9
Linear Complexity	5 000	1 115 562	5 950	187.5
Serial	9	28 811	106	271.4
Approximate Entropy	8	16 406	106	154.5
Cumulative Sums		287	24	11.9
Random Excursions		548	212	2.6
Random Excursions Variant		1 501	211	7.1
Total		1 317 407	26 038	50.6

4 Výsledky a ich interpretácia

V praxi sa testovanie náhodnosti realizuje tak, že sa vstupné dáta rozdelia na väčší počet (100 prípadne 1 000) postupností a na každú postupnosť sa aplikujú testy z batérie. Výsledkom tak je množina p-hodnôt (1 000), ktoré je potrebné vyhodnotiť pre každý test (podtest) zvlášť. Vyhodnotenie testu vychádza z faktu, že p-hodnoty sú pre náhodný generátor rovnomerne rozdelené na intervale [0,1]. Väčšina batérií rovnako ako NIST STS testuje rovnomernosť vypočítaných p-hodnôt. NIST STS naviac ešte vyhodnocuje náhodnosť na základe ďalšieho kritéria – celkový podiel postupností, ktoré boli vyhodnotené ako nenáhodné (p-hodnota$< \alpha$.) Rovnomernosť množiny p-hodnôt pre oba spôsoby vyhodnocuje s rovnakou hladinou významnosti ako $\alpha = 0{,}01$, teda dobrý generátor zamietame s pravdepodobnosťou 1 %.

4.1 Interpretácia výsledkov

Dôležitou súčasťou testovania náhodných generátorov je interpretácia výsledkov. Keďže dobrý generátor neprejde testom/podtestom s 1 % je možné teoreticky

stanoviť pravdepodobnosť, koľko nezávislých testov vyhlási generátor za nenáhodný. Problémom však je, že niektoré testy sú závislé a teda dobrý generátor „neprechádza“ väčším počtom testov ako je očakávané. Pre NIST STS s 188 testami/podtestami je teoretická pravdepodobnosť, že generátor prejde všetkými testami 15 %, zatiaľ čo skutočná pravdepodobnosť je 20 %. V práci [5] sme sa zamerali na analýzu počtu NIST STS testov, ktoré vyhlásia fyzikálny kvantový generátor za nenáhodný. Čiastočné výsledky možno vidieť v Tabuľka 3.

Tabulka 3: Teoretická a skutočná pravdepodobnosť, že kvantový generátor neprejde daným počtom testov (i)

počet testov (i)	0	1	2	3	4	5	6	7
očakávaná pravdepodobnosť	15,1	28,7	27,1	17,0	7,9	2,9	0,9	0,2
skutočná pravdepodobnosť	19,4	28,9	23,8	14,3	7,2	3,3	1,5	0,7

Z tabuľky plynie, že generátor možno považovať za náhodný, ak neprejde 7 alebo menej NIST STS testov. Ďalšie skutočnosti pre interpretáciu NIST STS testov možno nájsť v článku [5].

5 Záver

V článku sme v skratke popísali kľúčové časti a kroky štatistického testovania náhodných generátorov a dát. Predstavili sme tu optimalizovanú batériu testov NIST STS, ktorá je 50x rýchlejšia ako pôvodná verzia [1] a implementuje taktiež efektívnu (až 187× rýchlejšiu ako pôvodná implementácia) verziu Berlekamp-Massey algoritmu. V článku sme sa sčasti venovali tiež interpretácii výsledkov NIST STS, z ktorej plynie, že generátor možno považovať za náhodný keď neprejde 7 (alebo menej) testami z NIST STS.

Literatura

[1] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. NIST, 2001.
http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf

[2] Sýs, M., Říha, Z.: Faster Randomness Testing with the NIST Statistical Test Suite. In *Security, Privacy, and Applied Cryptography Engineering*, Rajat

Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont (Eds.). Lecture Notes in Computer Science, Vol. 8804. Springer, 272–284.

[3] Sýs, M., Říha, Z., Matyáš, V.: *Algorithm Berlekamp-Massey: Optimizing the NIST Statistical Test Suite*, ACM Transaction of Mathematical Software. Akceptovaný článok.

[4] Sýs, M., Říha, Z.: *Optimised implementation of NIST STS.* http://crcs.cz/projects/sts (2014–2016)

[5] Sýs, M., Říha, Z., Matyáš, V., Márton, K., Suciu, A.: *On the Interpretation of Results from the NIST Statistical Test Suite.* Romanian Journal of Information Science and Technology 18, 1, 2015.

[6] Suciu, A., Márton, K., Nagy, I., Pinca, I.: *Byte-oriented efficient implementation of the NIST statistical test suite.* International Conference on Automation, Quality and Testing, Robotics 2 (2010), 1–6.

[7] L'Ecuyer, P., Simard, R.: *TestU01: A C Library for Empirical Testing of Random Number Generators.* ACM Trans. Math. Softw. 33, 4, Article 22 (Aug. 2007).

[8] *FIPS PUB 140-2, Security Requirements for Cryptographic Modules, with Change Notices*, December 2002.

[9] Marsaglia, G.: *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness.* 1995.

The Million-Key Question – Investigating the Origins of RSA Public Keys

Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský, David Formánek, David Komárek, Vashek Matyáš

E-mail: svenda@fi.muni.cz

Abstract

Can bits of an RSA public key leak information about design and implementation choices such as the prime generation algorithm? We analysed over 60 million freshly generated key pairs from 22 open- and closed-source libraries and from 16 different smartcards, revealing significant leakage. The bias introduced by different choices is sufficiently large to classify a probable library or smartcard with high accuracy based only on the values of public keys. Such a classification can be used to decrease the anonymity set of users of anonymous mailers or operators of linked Tor hidden services, to quickly detect keys from the same vulnerable library or to verify a claim of use of secure hardware by a remote party. The classification of the key origins of more than 10 million RSA-based IPv4 TLS keys and 1.4 million PGP keys also provides an independent estimation of the libraries that are most commonly used to generate the keys found on the Internet.

Our broad inspection provides a sanity check and deep insight regarding which of the recommendations for RSA key pair generation are followed in practice, including closed-source libraries and smartcards.[1]

1 Introduction

The RSA key pair generation process is a crucial part of RSA algorithm usage, and there are many existing (and sometimes conflicting) recommendations regarding how to select suitable primes p and q [6, 8, 9, 10, 11] to be later used to compute the private key and public modulus. Once these primes have been

This is shortened version of paper which originally appeared on 25th Usenix Security Symposium, Texas, Austin, 2016.

[1] **This is a shortened version of paper which originally appeared on 25th Usenix Security Symposium, Austin, Texas, 2016.** Full details, paper supplementary material, datasets and author contact information can be found at http://crcs.cz/papers/usenix2016.

selected, modulus computation is very simple: $n = p \cdot q$, with the public exponent usually fixed to the value 65 537. But can the modulus n itself leak information about the design and implementation choices previously used to generate the primes p and q? Trivially, the length of the used primes is directly observable. Interestingly, more subtle leakage was also discovered by Mironov [12] for primes generated by the OpenSSL library, which unwantedly avoids small factors of up to 17 863 from $p-1$ because of a coding omission. Such a property itself is not a security vulnerability (the key space is decreased only negligibly), but it results in sufficiently significant fingerprinting of all generated primes that OpenSSL can be identified as their origin with high confidence. Mironov used this observation to identify the sources of the primes of factorizable keys found by [7]. But can the origins of keys be identified only from the modulus n, even when n cannot be factorized and the values of the corresponding primes are not known?

To answer this question, we generated a large number of RSA key pairs from 22 software libraries (both open-source and closed-source) and 16 different cryptographic smartcards from 6 different manufacturers, exported both the private and public components, and analysed the obtained values in detail. As a result, we identified seven design and implementation decisions that directly fingerprint not only the primes but also the resulting public modulus: 1) Direct manipulation of the primes' highest bits. 2) Use of a specific method to construct strong or provable primes instead of randomly selected or uniformly generated primes. 3) Avoidance of small factors in $p-1$ and $q-1$. 4) Requirement for moduli to be Blum integers. 5) Restriction of the primes' bit length. 6) Type of action after candidate prime rejection. 7) Use of another non-traditional algorithm – functionally unknown, but statistically observable.

As different design and implementation choices are made for different libraries and smartcards (cards) with regard to these criteria, a cumulative fingerprint is sufficient to identify a probable key origin even when only the public key modulus is available. The average classification accuracy on the test set was greater than 73 % even for a single classified key modulus when a hit within the top 3 matches was accepted[2]. When more keys from the same (unknown) source were classified together, the analysis of as few as ten keys allowed the correct origin to be identified as the top single match in more than 85 % of cases. When five keys from the same source were available and a hit within the top 3 matches was accepted, the classification accuracy was over 97 %.

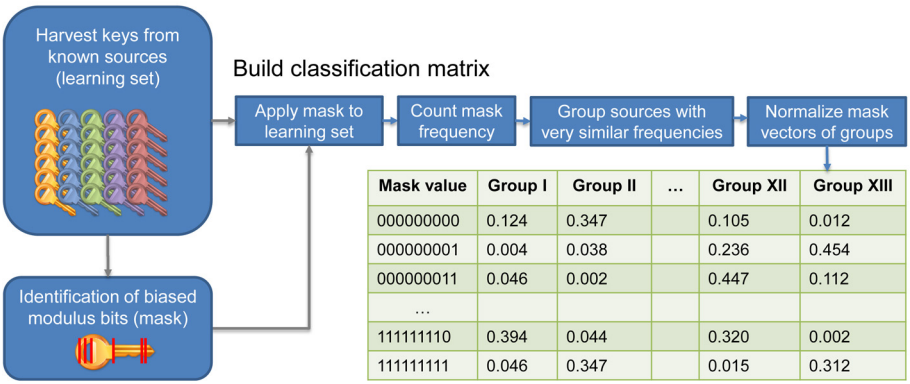
We also used the proposed probabilistic classifier to classify RSA keys [14] collected from the IPv4 HTTPS/TLS [4], Certificate Transparency [5] and PGP [15] datasets and achieved remarkably close match to the current market share of web servers for TLS dataset.

[2]The correct library is listed within the first three most probable groups of distinct sources identified by the classification algorithm.

2 Key source detection

The distinct distributions of specific bits of primes and moduli enable probabilistic estimation of the source library or card from which a given public RSA key was generated. Intuitively, classification works as follows:

1. Bits of moduli known to carry bias are identified with additional bits derived from the modulus value (a *mask*, $6 + 3$ bits in our method).
2. The frequencies of all possible *mask* combinations (2^9) for a given source in the learning set are computed as shown on Figure 1.



Mask value	Group I	Group II	...	Group XII	Group XIII
000000000	0.124	0.347		0.105	0.012
000000001	0.004	0.038		0.236	0.454
000000011	0.046	0.002		0.447	0.112
...					
111111110	0.394	0.044		0.320	0.002
111111111	0.046	0.347		0.015	0.312

Fig. 1: The first phase of classification of the RSA public keys. The groups of similar sources are established based on the statistical properties of a large number of keypairs generated by separate sources. As a result, a fixed classification matrix is pre-computed

3. For classification of an unknown public key, the bits selected by the *mask* are extracted as a particular value v. The source with the highest computed frequency of value v (step 2) is identified as the most probable source. When more keys from the same source are available (multiple values v_i), a higher classification accuracy can be achieved through element-wise multiplication of the probabilities of the individual keys. Both cases are shown on Figure 2.

The average accuracy on the test set of the most probable source group was over 40 % for single keys and improved to greater than 93 % when we used batches of 100 keys from the same source for classification [14].

When 10 keys from the same source were classified in a batch, the most probable classified group was correct in more than 85 % of cases and was almost always (99 %) included in the top three most probable sources. A significant

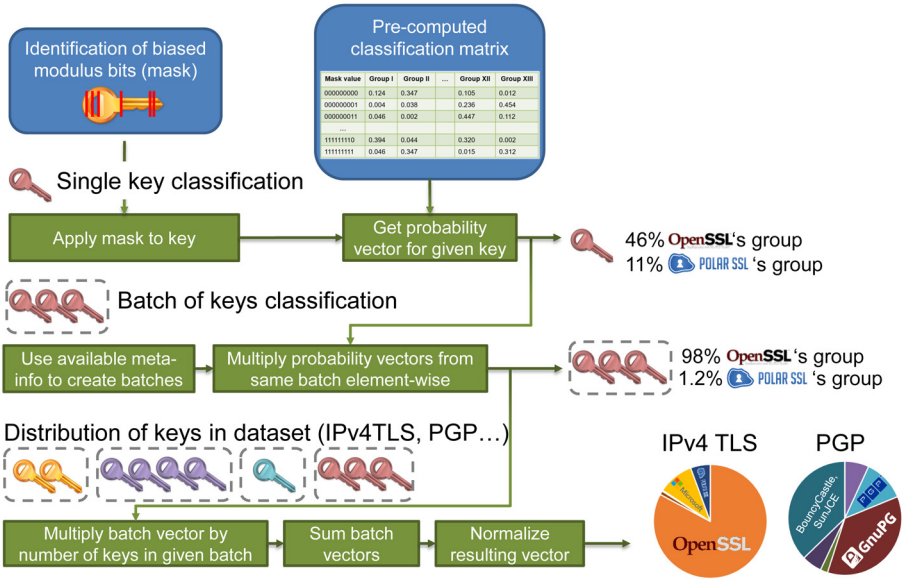


Fig. 2: An example of the second phase of classification. Either a) single key, b) batch of keys from the same source or c) whole dataset of batches is classified using pre-computed classification matrix

variability in classification success was observed among the different groups. E.g., G&D smartcards cards and PGPSDK4 FIPS library could be correctly identified from even a single key because of their distinct distributions of possible mask values. By contrast Microsoft providers (CryptoAPI, CNG and .NET) were frequently misclassified when only a single key was used because of the wider range of possible mask values, resulting in a lower probability of each individual mask value.

We conclude that our classification method is moderately successful even for a single key and very accurate when a batch of at least 10 keys from the same source is classified simultaneously.

To ease the classification process, we created online tool which classifies RSA public key extracted from provided certificate in ASCII armor or TLS handshake of given server's URL. The interface of the tool is shown at Figure 3.

Further leakage in other bits of public moduli might be found by applying machine learning methods to the learning set, potentially leading to an improvement of the classification accuracy. Moreover, although we have already tested a wide range of software libraries and cards, more sources could also be incorporated, such as additional commercial libraries, various hardware security modules and additional types of cards and security tokens.

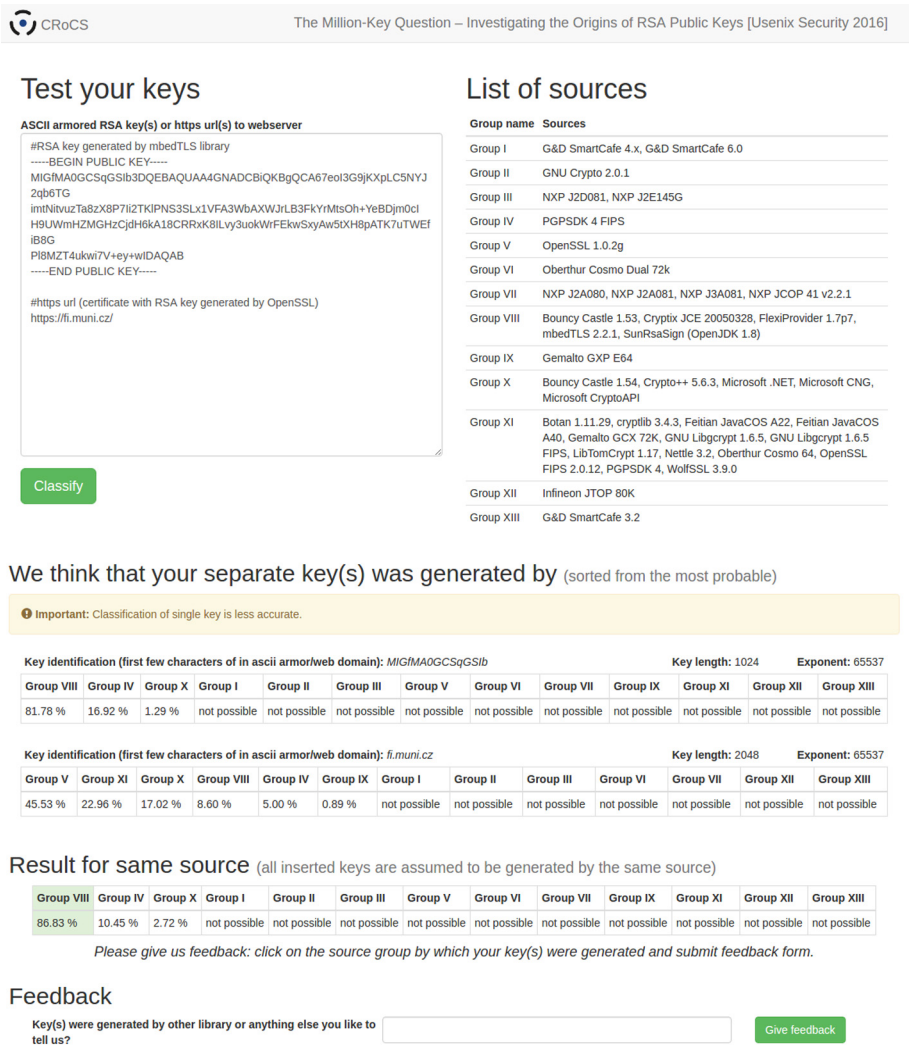


Fig. 3: Screenshot of our online classification tool, available at http://crcs.cz/rsapp. The website supports ASCII armored RSA keys and retrieval of RSA keys found in TLS certificate of supplied domain name. The keys are classified individually and as a group assumed to be generated by the same source

2.1 Practical impact of origin detection

The possibility of accurately identifying the originating library or card for an RSA key is not solely of theoretical or statistical interest. If some library or card is found to produce weak keys, then an attacker can quickly scan for other keys from the same vulnerable source. The possibility of detection is especially helpful when a successful attack against a weak key requires a large but practically achievable amount of computational resources. Preselecting potentially vulnerable keys saves an attacker from spending resources on all public keys.

The identification of the implementations responsible for the weak keys found in [2, 7] was a difficult problem. In such cases, origin classification can quickly provide one or a few of the most probable sources for further manual inspection. Additionally, a set of already identified weak keys can be used to construct a new classification group, which either will match an already known one (for which the underlying sources are known) or can be used to search for other keys that belong to this new group in the remainder of a larger dataset (even when the source is unknown).

Another practical impact is the decreased anonymity set of the users of a service that utilizes the RSA algorithm whose users are not intended to be distinguishable (such as the Tor network). Using different sources of generated keys will separate users into smaller anonymity groups, effectively decreasing their anonymity sets. The resulting anonymity sets will be especially small when individual users decides to use cryptographic hardware to generate and protect their private keys (if selected device does not fall into into group with widely used libraries). Note that most users of the Tor project use the default client, and hence the same implementation, for the generation of the keys they use. However, the preservation of indistinguishability should be considered in the development of future alternative clients.

Tor hidden services sometimes utilize ordinary HTTPS certificates for TLS [1], which can be then linked (via classification of their public keys) with other services of the same (unknown) operator.

Mixnets such as mixmaster and mixminion use RSA public keys to encrypt messages for target recipient and/or intermediate mix. If key ID is preserved, one may try to obtain corresponding public key from PGP keyserver and search for keys with the same source to narrow that user's anonymity set in addition to analysis like one already performed on alt.anonymous.messages [13]. Same as for Tor network, multiple seemingly independent mixes can be linked together if uncommon source is used to generate their's RSA keys.

A related use is in a forensic investigation in which a public key needs to be matched to a suspect key-generating application. Again, secure hardware will more strongly fingerprint its user because of its relative rarity.

An interesting use is to verify the claims of remote providers of Cryptography as a Service [3] regarding whether a particular secure hardware is used as claimed. As the secure hardware (cards) used in our analysis mostly exhibit distinct properties of their generated keys, the use of such hardware can be distinguished from the use of a common software library such as OpenSSL.

2.2 How to mitigate origin classification

The impact of successful classification can be mitigated on two fronts: by library maintainers and by library users. The root cause lies with the different design and implementation choices for key generation that influence the statistical distributions of the resulting public keys. A maintainer can modify the code of a library to eliminate differences with respect to the approach used by all other sources (or at least the most common one, which is OpenSSL in most cases). However, although this might work for one specific library (mimicking OpenSSL), it is not likely to be effective on a wider scale. Changes to all major libraries by its maintainers are unlikely to occur, and many users will continue to use older versions of libraries for legacy reasons.

More pragmatic and immediate mitigation can be achieved by the users of these libraries. A user may repeatedly generate candidate key pairs from his or her library or device of choice and reject it if its classification is too successful. Expected number of trials differs based on the library used and the prior probability of sources within the targeted domain. For example, if TLS is the targeted domain, five or less key generation trials are expected for most libraries to produce "indecisive" key.

The weakness of the second approach lies in the unknown extent of public modulus leakage. Although we have described seven different causes of leakage, others might yet remain unknown – allowing for potential future classification of keys even after they have been optimized for maximal indecisiveness against these seven known causes.

This strategy can be extended when more keys are to be generated. All previously generated keys should be included in a trial classification together with the new candidate key. The selection process should also be randomized to some extent; otherwise, a new classification group of "suspiciously indecisive" keys might be formed.

3 Conclusions

This paper describes the possibility to identify a source of RSA key(s) based on the knowledge of a public key only. During the key generation process, the source-specific prime selection algorithms, postprocessing techniques and en-

forcement of specific properties (e.g., Blum primes) make the resulting primes slightly biased, and these biases serve as fingerprints of the sources. Our paper therefore shows that public moduli leak significantly more information than previously assumed. We identified seven properties of the generated primes that are propagated into the public moduli of the generated keys. As a result, accurate identification of originating library or smartcard is possible based only on knowledge of the public keys. Such an unexpected property can be used to decrease the anonymity set of RSA keys users, to search for keys generated by vulnerable libraries, to assess claims regarding the utilization of secure hardware by remote parties, and for other practical uses. We classified the probable origins of keys in two large datasets consisting of 10 and 15 million (mostly) TLS RSA keys and 1.4 million PGP RSA keys to obtain an estimate of the sources used in real-world applications.

Acknowledgements

We acknowledge the support of the Czech Science Foundation, project GA16-08565S. The access to the computing and storage resources of National Grid Infrastructure MetaCentrum (LM2010005) is greatly appreciated. We would like to thank all anonymous reviewers and our colleagues for their helpful comments and fruitful discussions.

References

[1] Arma: *Tor blog: Facebook, hidden services, and https certs.* Available from https://blog.torproject.org/blog/facebook-hidden-services-and-https-certs (cit. 26. 06. 2016)

[2] Bernstein, D. J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., Someren, N.: Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *Advances in Cryptology – ASIACRYPT 2013*. Springer-Verlag, 2013, pp. 341–360.

[3] Berson, T., Dean, D., Franklin, M., Smetters, D., Spreitzer, M.: Cryptography as a network service. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2001.

[4] Durumeric, Z., et al.: *Internet-Wide Scan Data Repository: Full IPv4 HTTPS Handshakes, dump from June 02, 2016.*
Available from: https://scans.io/ (cit. 02. 06. 2016)

[5] Google: *Certificate Transparency dump from June 07, 2016.*
https://www.certificate-transparency.org (cit. 07. 06. 2016)

[6] Gordon, J.: *Strong Primes are Easy to Find.* Springer-Verlag, 1985, pp. 216–223.

[7] Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J. A.: Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices, In *21st USENIX Security Symposium. Proceedings.* USENIX, 2012, pp. 205–220.

[8] IEEE: *Standard Specifications for Public-Key Cryptography.* IEEE Std 1363, 2000.

[9] Kerry, C. F., Romine, C.: *FIPS PUB 186-4 Digital Signature Standard (DSS)*, 2013.

[10] Loebenberger, D., Nüsken, M.: Notions for RSA Integers. In *International Journal of Applied Cryptography.* Inderscience Publishers, 2014, pp. 116–138.

[11] Maurer, U. M.: Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology 8*, 3 (1995), 123–155.

[12] Mironov, I.: *Factoring RSA Moduli II.* Available from https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/ (cit. 26. 06. 2016)

[13] Ritter, T.: *De-anonymizing alt.anonymous.messages.* Available from https://ritter.vg/p/AAM-defcon13.pdf (cit. 26. 06. 2016)

[14] Švenda, P., Nemec, M., Sekan, P., Kvašňovský, R., Formánek, D., Komárek, D., Matyáš, V.: The Million-Key Question – Investigating the Origins of RSA Public Keys. In *25th USENIX Security Symposium. Proceedings*, 2016.

[15] *PGP keydump from June 02, 2016.* Available from: http://pgp.key-server.io/dump/current/ (cit. 02. 06. 2016)

RSA key generation in cryptographic libraries

Matúš Nemec

E-mail: mnemec@mail.muni.cz

Abstract

This article focuses on RSA key generation algorithms used by open-source cryptographic libraries. The differences in described methods influence statistical properties of the produced keys in such significant way that many popular libraries are distinguishable from just a small sample of generated public keys. Such information leakage vulnerability may have negative implications for anonymity of Internet users. Additional properties of private keys reveal details of the implementation, which allows for reverse engineering of proprietary solutions.

1 Introduction

Key pairs generated for the RSA cryptosystem are often used for long periods of time. When RSA is chosen for TLS or secure email, a compromise of the private key leads to a loss of confidentiality of all past and future communication captured by an attacker. The correct choice of the RSA parameters is of utmost importance for the overall security of the application.

Many standards and publications concerning RSA exist; however, few give a precise algorithm for key generation. Across different publications, the requirements placed on the RSA key pairs are often contradictory. We surveyed 18 widely used cryptographic libraries and compared their algorithms for key generation. While most of them adhere to at least some international standard (mostly due to weak requirements set by some standards), several implementations add their own peculiarities to the process.

Fortunately, despite wide variety of algorithms, none of them break security of RSA, yet they are characterized by a lot of interesting statistical properties. Some can be very common or hardly observable, however a few of them uniquely fingerprint keys as coming from a specific library.

When properties of algorithms are observed individually, there are often too few differences. However, the libraries use algorithms in various combinations. Additionally, bugs and uncommon implementation choices contribute to the variety of libraries.

2 RSA key generation

A key pair for the RSA cryptosystem requires two large primes p and q. The public key consist of the modulus $n = p \cdot q$ and the public exponent e (often $e = 2^{16} + 1 = 65\,537$). The private exponent d may be computed as modular inverse of the public exponent ($d = e^{-1} \bmod \phi(n)$, $\phi(n)$ is the Euler's totient function; $\phi(n) = (p-1) \cdot (q-1)$), assuming e and $\phi(n)$ are coprime ($GCD(e, \phi(n)) = 1$).

The RSA key size is the bit length of the modulus n. Typically, the key size is selected first and then two primes of appropriate lengths are generated. There are several methods for generating large primes. Additionally, pairs of primes can be also selected in different ways. The primes may be ordered after they are generated (either $p > q$ or $p < q$).

Usually, the size of each prime is exactly half of the key size. However, multiple of such primes does not always have the desired bit length. This inspires the notion of the *maximal region* for RSA moduli (Figure 1), defined by precise length of the modulus and precise length of the primes. There are two principal ways how to solve the problem of modulus length. Both methods have characteristic distributions of primes and moduli.

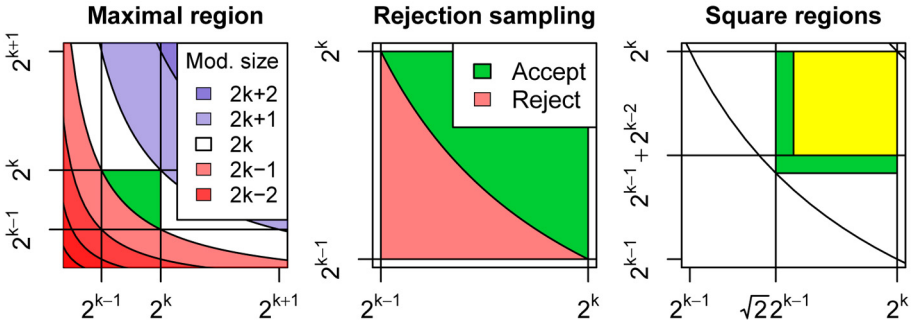


Fig. 1: Regions for generating prime pairs. **Left:** bit length of the modulus $n = p \cdot q$ depends on the values of primes p and q. Typically, the primes must have bit length k and the modulus must be $2k$ bits long. The conditions are satisfied by the green region. **Middle:** generating RSA keys from the maximal region. Primes are generated from the interval $[2^{k-1}, 2^k - 1]$. A pair of primes that does not produce modulus of correct length (short by one bit) is rejected. **Right:** Generating RSA keys without rejection sampling. The primes are chosen from the interval $[\sqrt{2} \cdot 2^{k-1}, 2^k - 1]$ for the larger square and $[3 \cdot 2^{k-2}, 2^k - 1]$ for the smaller square. The latter region can be achieved by fixing the two most significant bits of primes to 11_2

2.1 Rejection sampling

Generating primes from distributions other than uniform is impractical. To overcome this issue, the primes are generated uniformly from the interval $[2^{k-1}, 2^k - 1]$. If their multiple does not have correct length, the process is repeated. Hence, the RSA keys are generated from the maximal region.

The primes should be selected independently, two new primes should be generated every time the pair is from the rejected region (*mbedTLS, FlexiProvider, Cryptix*). If the first prime is generated uniformly, and the second prime is generated until the modulus has correct length, the distribution of moduli is clearly biased on small values (*GNU Crypto*). If the algorithm preserves the greater prime and only replaces the lesser one, the distribution is still slightly biased (*OpenJDK/SunRsaSign, older Bouncy Castle*).

2.2 Practical intervals

To avoid the need to generate new primes, they are selected from intervals that produce the correct modulus length. Such choices are characterized by square regions in the (p, q)-plane (Figure 1). The simplest such region is achieved by fixing the two most significant bits of primes to 11_2. Hence the primes are selected from the interval $[3 \cdot 2^{k-2}, 2^k - 1]$ (*Botan, Apple corecrypto, cryptlib, libgcrypt, LibTomCrypt, Nettle, OpenSSL, PGP SDK 4, WolfSSL*).

Alternatively, the largest square region is achieved when generating from the intervals $[\sqrt{2} \cdot 2^{k-1}, 2^k - 1]$ (*ANSI X9.31, FIPS 186-4, RSAES-OAEP specification, newer Bouncy Castle, Crypto++, MS CryptoAPI*). Such bounds for prime generation can be achieved by rejection sampling. If a negligible bias is acceptable, then it can be performed more efficiently, with one call to a random number generator [16].

3 Prime generation

There are three main methods for generating large primes. Random numbers (or numbers from a sequence) can be tested with probabilistic tests of compositeness, yielding *probable primes*. *Provably prime* numbers are constructed such that their primality can be reasoned mathematically. *Primes with conditions* are generated to satisfy certain congruence conditions. For an analysis beyond the scope of this section, please refer to [13].

3.1 Probable primes

True tests of primality either require large computational effort (Jacobi sum test, elliptic curve primality proving, partial factorization of $p - 1$) or work only for numbers from special classes (Mersenne numbers) [9]. Since RSA requires large primes of general form, random numbers are instead tested for primality using

knowledge from number theory. Since the probabilistic compositeness tests are always correct when they declare a number to be composite, no prime numbers are eliminated. The answer that a number is prime is uncertain. The test is repeated with different parameters to increase the confidence in the answer *probable prime*. The software libraries use various combinations of the Fermat [9], Solovay-Strassen [17], Miller-Rabin [10, 14] and Lucas test [1].

Random sampling Random candidates are generated uniformly until a prime is found (*NIST FIPS 186-4, IEEE 1363-2000, Apple corecrypto, GNU Crypto, LibTomCrypt, WolfSSL*). The compositeness test can be made more efficient if the candidate is first tested by trial division with a few small primes. Alternatively, a GCD computation with a product of a few primes may reveal several prime divisors at once.

Incremental search A random starting point is selected and then incremented until it is prime (*RSAES-OAEP specification, Botan, Bouncy Castle, Cryptix, cryptlib, Crypto++, FlexiProvider, mbedTLS, OpenJDK, OpenSSL, PGP SDK 4*). Small bias exist and is caused by different sizes of "gaps" between prime numbers. The method can be sped up using trial division, sieve of Eratosthenes or with a continuously updated table of remainders.

3.2 Provable primes

Large prime numbers are recursively constructed from smaller primes using Pocklington's theorem [9]. The process is randomized, but the primality of the outcome can be proven. The algorithm was first proposed by Maurer [8]. It is used by *Nettle*.

3.3 Primes with conditions

A *strong prime* p [15] is generated in such way, that $p-1$ has a large prime factor p^-, $p+1$ has a large prime factor p^+ and optionally, $p^- - 1$ has a large prime factor p^{--}. The conditions will guarantee that the key is safe from cycling attacks [2] and *Pollard's* $p-1$ and *Williams'* $p+1$ factorization method [15]. Such primes can be produced uniformly (*ANSI X9.31, NIST FIPS 186-4, corecrypto, IEEE 1363-2000, libgcrypt, OpenSSL FIPS*) or with a bias (Gordon's algorithm in *PGP SDK 4 FIPS*).

4 Statistical properties of RSA keys

There is a difference in how much information can be gathered about a library from its RSA keys, depending on the type of keys available. In most scenarios, only public keys will be accessible (e.g., keys gathered from public key certificates). Naturally, private keys provide more information. The most detailed analysis requires additional computation with the private keys (e.g., factoriza-

tion of $p-1$ and $p+1$, where p is a private prime). The analysis of public keys may help to de-anonymize Internet users [18]. The detailed processing of private keys reveals details about proprietary implementations, where source code is not a available, as we demonstrate by analysing the keys generated with Microsoft libraries.

4.1 Public modulus

A histogram of a few most significant bits of the public modulus value reveals the *algorithm for prime pair selection* as described in section 2. It is possible to distinguish all types of prime pair regions. With sufficient number of keys, the different approaches to rejection sampling are also clearly visible [13]. However, different combinations of prime generation intervals may produce the same distribution of the modulus (e.g., it is not possible to see from modulus if the primes are ordered or not).

The use of *Blum integers* can be detected from the moduli with a high precision, as random moduli are equiprobably congruent to 1 and 3 modulo 4, whereas Blum integers are always congruent to 1 modulo 4. The probability that k random moduli will be Blum integers is 2^{-k}. Only *GNU Crypto* uses such moduli, caused by a bug [13].

The OpenSSL library has a unique property amongst all the examined libraries. *Non-randomness in small factors of* $p-1$ can be detected when sufficiently many keys are available. Specifically, candidates for p for which $p-1$ is divisible by a small prime up to 17 863 (except 2) are rejected. As seen in table 1, n is congruent to 1 more often than to other values. It follows from Dirichlet's theorem that the remainders should be distributed uniformly among the $\phi(n)$ congruence classes in $\mathbb{Z}_n^*$ [9, Fact 4.2].

Table 1: Probability of remainders modulo small primes for moduli generated by OpenSSL. Remainders of random moduli are uniformly distributed

	Remainder													
Divisor	1	2	3	4	5	6	...	10	11	12	...	16	...	18
3	1	0												
5	3/9	2/9	2/9	2/9										
7	5/25	4/25	4/25	4/25	4/25	4/25								
11	9/81	8/81	8/81	8/81	8/81	8/81	...	8/81						
13	11/121	10/121	10/121	10/121	10/121	10/121	...	10/121	10/121	10/121				
17	15/225	14/225	14/225	14/225	14/225	14/225	...	14/225	14/225	14/225	...	14/225		
19	17/289	16/289	16/289	16/289	16/289	16/289	...	16/289	16/289	16/289	...	16/289	...	16/289
						...								
17 863	$\frac{1}{17\,861}$	$\frac{17\,860}{17\,861^2}$	...	...	...	...	...	...	...	...	...	...	...	...
17 881	1/17 880	1/17 880	...	...	...	...	...	...	...	...	...	...	...	...
17 891	1/17 890	1/17 890	...	...	...	...	...	...	...	...	...	...	...	...

1. Generate random odd candidate c of $nLen/2$ bits.
2. While c or $c-1$ is divisible by $p_1 \dots p_{2048}$, $c = c+2$.
3. If c is not a probable prime, go to 1.
4. If $(c-1)/2$ is not a probable prime, go to 1.
5. Output a safe prime c.

1. Generate random odd candidate c of $nLen/2$ bits.
2. While c or $c-1$ is divisible by $p_2 \dots p_{2048}$, $c = c+2$.
3. If c is not a probable prime, go to 1.
4. ~~If $(c-1)/2$ is not a probable prime, go to 1.~~
5. Output an OpenSSL prime c.

Fig. 2: Generating safe primes (left). The algorithm resembles generating primes in the OpenSSL library (right), however the primality of $(p-1)/2$ is not required by OpenSSL

Rejecting candidates p with small divisors of $(p-1)/2$ is useful to speed up the search for a *safe prime*; however, $(p-1)/2$ is not required to be prime by OpenSSL (see figure 2). This strange behaviour was first reported by Mironov [11] and seems to be caused by a bug, rather than by design.

4.2 Private keys – primes

The access to private keys reveals the *precise prime generation intervals*, as seen in figures 3, 4. Especially, it is possible to check if the primes are ordered ($p > q$ or $p < q$) or not.

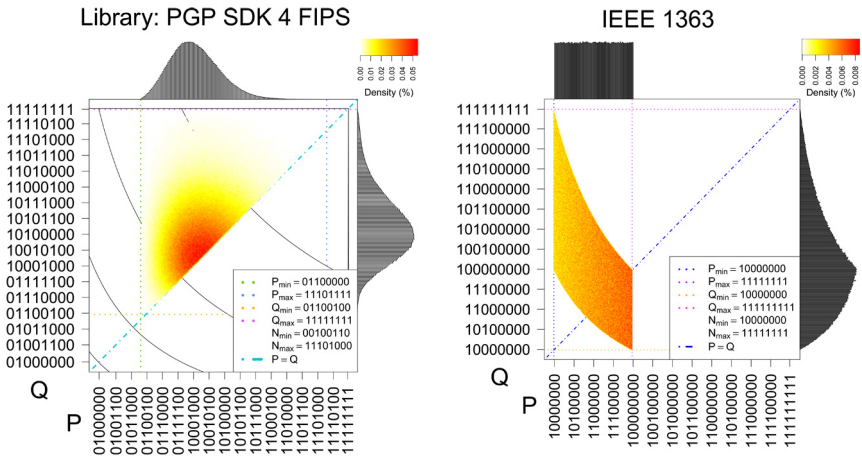


Fig. 3: Example distributions of primes. The histograms on the top and the side of the graph represent the marginal distributions of p and q, respectively. The colour scheme expresses the likelihood that primes of a randomly generated key will have specific high order bytes, ranging from white (not likely) over orange to red (more likely). The distributions of PGP SDK 4 in FIPS mode and IEEE 1363-2000 are very unique

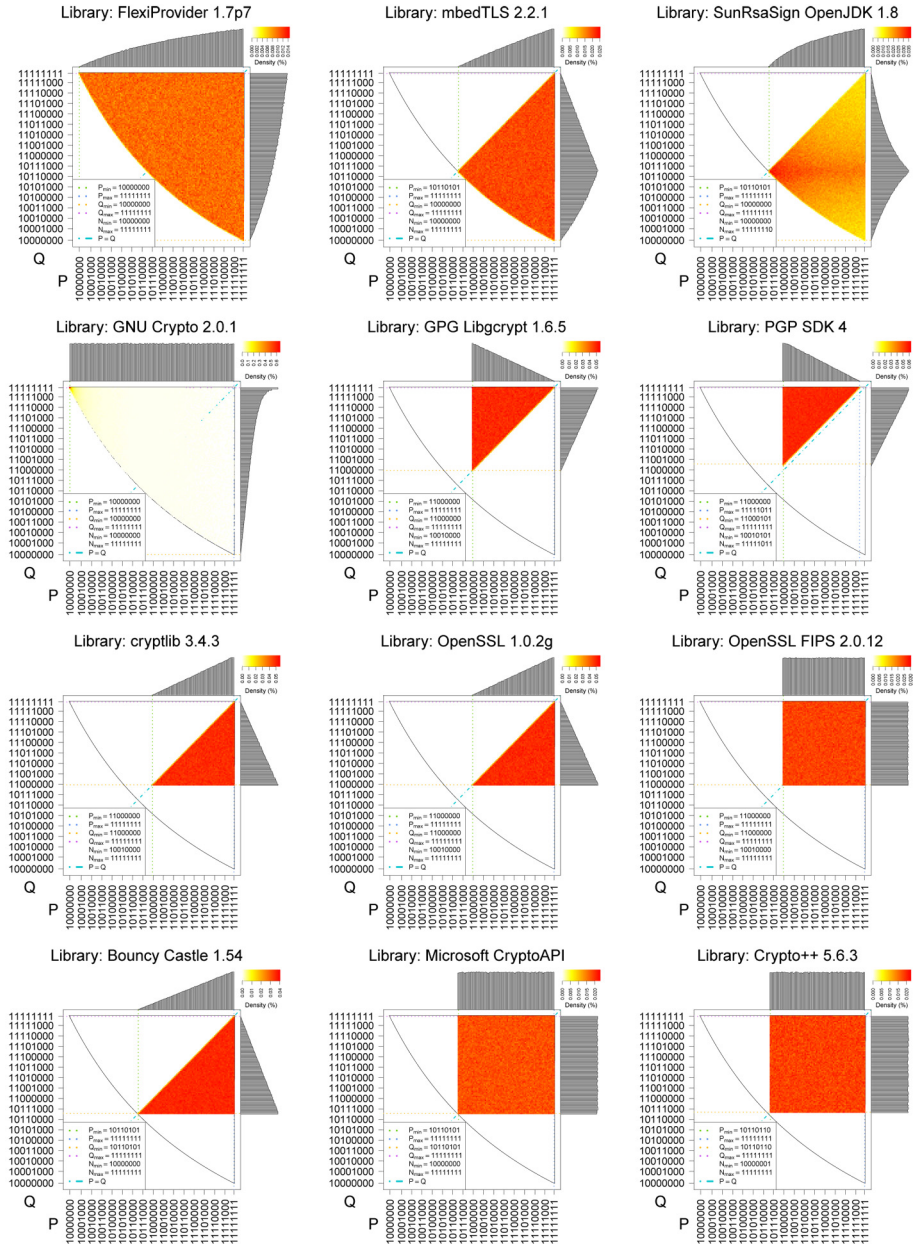


Fig. 4: Observed distributions of prime pairs in various libraries. The regions make it possible to distinguish different groups of libraries. Two uncommon distributions can be found in figure 3, where the graphs are also explained

Avoidance of small factors of $p-1$ is directly observable, since the property is no longer obfuscated by multiplication of the primes. The primes will never be congruent to 1 modulo any prime which is being avoided by OpenSSL (3 to 17 863).

The *minimal size of the difference* $p-q$ may be seen. *PGP SDK 4* requires the primes to differ somewhere in their 6 most significant bits. Many libraries check if the primes differ somewhere in their highest 100 bits. The opposite happens with very small probability (2^{-100}), hence it is not detectable in such case.

4.3 Factorization of private parameters

Once the primes p and q are known, it is possible to factorize the values $p-1$ and $p+1$. *The type of prime – random, provable or strong* (as described in section 3) can be then deduced from the sizes of the two largest prime factors. The differences are illustrated by figure 5. This part of the analysis is the most computationally expensive and is difficult to automate. The factorized values were provided by Rudolf Kvašňovský [18].

The two largest factors of $p-1$ and $p+1$, where p is a *random (probable) prime*, are randomly distributed.

When *strong primes* are used, $p-1$ and $p+1$ must be guaranteed to have at least one large factor. Its binary length will be bounded from the bottom, possibly even fixed to a certain length.

Provable primes will have factors of $p-1$ of specific size (approximately $\sqrt{p}$ for Maurer's algorithm or $\sqrt[3]{p}$ for an improved version thereof), while the factors of $p+1$ will be distributed randomly.

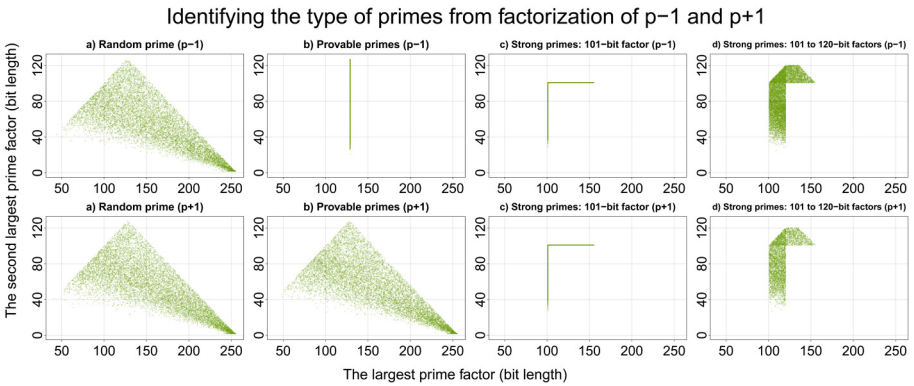


Fig. 5: Identifying the type of primes based on factorization of $p-1$ and $p+1$

4.4 Rare algorithms

Some algorithms differ in such significant way, that they are uniquely identifiable from just a subset of the properties.

PGP SDK 4 in FIPS mode often generates primes, which are 1 bit shorter than half of the modulus length specified by the user. As a result, the keys may be 1 or 2 bits shorter than desired. Additionally, the distributions of primes and moduli are very unique and can be easily distinguished when compared to other examined software libraries (figure 3).

The *IEEE 1363-2000* standard proposes a key generation algorithm, which produces almost uniformly distributed moduli. However, the downside is that one of the primes may be one bit larger than half of the modulus (figure 3). Small biases in modulus distribution do not seem to pose security risks [7]. Uniform distribution is therefore not necessary and because of the disadvantage, no library uses the proposed algorithm.

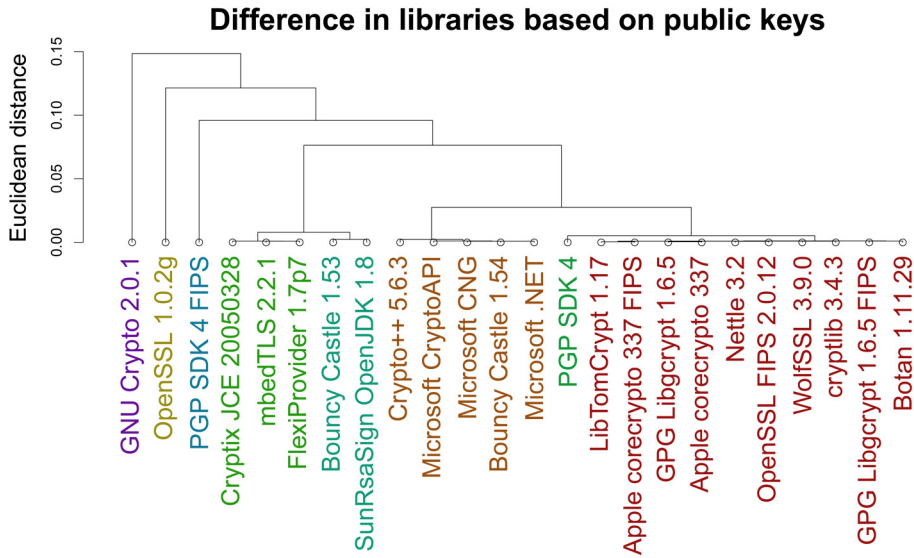


Fig. 6: Comparison of the libraries based on public keys only. The dendrogram is generated based on the frequencies of the following parts of the public keys: the most significant byte of the modulus, the 2^{nd} least significant bit of the modulus, modulus modulo 3

Table 2: Comparison of cryptographic libraries and standards. **Prime search method:** incremental search (Inc.); random sampling (Rand.); FIPS 186-4 Appendix B.3.6 or equivalent algorithm from ANSI X9.31 (FIPS); Maurer's algorithm (Maurer); PGP strong primes (PGP); IEEE 1363 strong primes (IEEE$^+$). **Prime pair selection:** rejection sampling (RS); primes with the two most significant bits 11_2 (11_2); primes from the interval $[\sqrt{2} \cdot 2^{\frac{n}{2}-1}, 2^{\frac{n}{2}} - 1]$ ($\sqrt{2}$); IEEE 1363 algorithm (IEEE). **Ordered primes:** $p > q$ (p); $p < q$ (q); random (×). **Fixed size of factors:** large factors of $p \pm 1$ have fixed length (√); large factors of $p \pm 1$ should differ in length (×); does not apply (–)

Library/Standard	Version	Prime search method	Prime pair selection	Large factor of $p-1$	Large factor of $p+1$	Ordered primes	Fixed size of factors	Notes
Apple corecrypto	337	Rand.	11_2	×	×	×	–	
Apple corecrypto	337 FIPS	FIPS	11_2	√	√	p	√	(p^-, p^+)-*strong* primes
Botan	1.11.29	Inc.	11_2	×	×	×	–	
Bouncy Castle	1.53 (Java)	Inc.	RS	×	×	p	–	Rejection sampling is less biased
Bouncy Castle	1.54 (Java)	Inc.	$\sqrt{2}$	×	×	p	–	
Cryptix JCE	20050328	Inc.	RS	×	×	×	–	Rejection sampling is not biased
cryptlib	3.4.3	Inc.	11_2	×	×	p	–	
Crypto++	5.6.3	Inc.	$\sqrt{2}$	×	×	×	–	255 ≥ prime MSB ≥ 182 = $\lceil\sqrt{2} \cdot 128\rceil$
FlexiProvider	1.7p7	Inc.	RS	×	×	×	–	Rejection sampling is not biased
GNU Crypto	2.0.1	Rand.	RS	×	×	×	–	Rejection sampling is more biased
GPG Libgcrypt	1.6.5	Inc.	11_2	×	×	q	–	Used by GnuPG 2.0.30
GPG Libgcrypt	1.6.5 FIPS	FIPS	11_2	√	√	q	√	FIPS 140-2 mode
LibTomCrypt	1.17	Rand.	11_2	×	×	×	–	
mbedTLS	2.2.1	Inc.	RS	×	×	p	–	Rejection sampling is not biased
Microsoft CryptoAPI	Windows 10	FIPS?	$\sqrt{2}$	√	√	×	×	Identical to MS CNG and MS .NET
Nettle	3.2	Maurer	11_2	√	×	×	√	Provable primes
OpenSSL	1.0.2g	Inc.	11_2	×	×	p	–	No small factors of $(p-1)/2$
OpenSSL FIPS	2.0.12 FIPS	FIPS	11_2	√	√	×	√	(p^-, p^+)-*strong* primes
PGP SDK 4.x	Desktop 10.0.1	Inc.	11_2	×	×	q	–	
PGP SDK 4.x	FIPS mode	PGP	11_2	√	√	q	√	(p^-, p^+, p^{--})-*strong* primes
SunRsaSign Provider	OpenJDK 1.8	Inc.	RS	×	×	p	–	Rejection sampling is less biased
WolfSSL	3.9.0	Rand.	11_2	×	×	×	–	
NIST FIPS 186-4 [12]	2013	FIPS	$\sqrt{2}$	√	√	×	×	Provable or (p^-, p^+)-*strong* primes
NIST FIPS 186-4	2013	Rand.	$\sqrt{2}$	×	×	×	–	Probable primes for 2048-bit keys
ANSI X9.31 / X9.44 [4]	1998 / 2007	FIPS	$\sqrt{2}$	√	√	×	×	(p^-, p^+)-*strong* primes
IEEE 1363 [3]	2000	IEEE$^+$	IEEE	√	√	×	×	(p^-, p^+, p^{--})-*strong* primes
IEEE 1363	2000	Rand.	IEEE	×	×	×	–	Probable primes allowed
RSAES-OAEP [16]	2000	Inc.	$\sqrt{2}$	×	×	×	–	
ISO/IEC 18033-2 [5]	2006	–	–	×	×	×	–	Only necessary conditions
PKCS #1 [6]	2.1 (RFC 3447)	–	–	×	×	×	–	Only necessary conditions

5 Conclusions

We have shown that the RSA key generation techniques used by open-source libraries leak information into the keys. The amount of information depends on the type of keys available. Many libraries seem to be identical when comparing their public keys. However, access to private keys and to other derived information uncovers more information, up to completely revealing the used algorithms. The results do not seem to create any serious security vulnerability.

The methodology can be used to track changes between versions (as is the case of the Bouncy Castle library) or to check standard compliance (keys produced by a tested library should be unrecognisable from a reference implementation with the highest level of detail). In case of key classification as performed in [18], the comparison based on public keys will unveil which libraries are not distinguishable.

References

[1] Berrizbeitia, P., Berry, T. G.: Generalized Strong Pseudoprime Tests and Applications. *J. Symb. Comput.*, 30(2):151–160, August 2000.

[2] Gysin, M., Seberry, J.: Generalised Cycling Attacks on RSA and Strong RSA Primes. In *Information Security and Privacy: ACISP'99. Proceedings*, pages 149–163. Springer-Verlag, 1999.

[3] IEEE: *Standard Specifications for Public-Key Cryptography.* IEEE Std 1363, 2000.

[4] American National Standards Institute: *ANSI X9.31-1998: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA)*, 1998.

[5] International Organization for Standardization: *Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers.* ISO/IEC 18033-2, 2006.

[6] Jonsson, J., Kaliski, B.: *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.* RFC 3447, 2003.

[7] Loebenberger, D., Nüsken, M.: Notions for RSA Integers. In *International Journal of Applied Cryptology*, pages 116–138. Inderscience Publishers, 2014.

[8] Maurer, U. M.: Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

[9] Menezes, A. J., Van Oorschot, P. C., Vanstone, S. A., Rivest, R. L.: *Handbook of Applied Cryptography*. CRC Press, 1st edition, 1996.

[10] Miller, G. L.: Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[11] Mironov, I.: *Factoring RSA Moduli. Part II*, 2012. Online: https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/ (cit. 01. 09. 2016)

[12] National Institute of Standards and Technology: *Digital Signature Standard (DSS)*. FIPS 186-4, 2013.

[13] Nemec, M.: *The properties of RSA key generation process in software libraries*. Master's thesis, 2016. Online: https://is.muni.cz/th/396066/fi_m/ (cit. 01. 09. 2016)

[14] Rabin, M. O.: Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[15] Rivest, R. L., Silverman, R. D.: Are 'Strong' Primes Needed for RSA? In *The 1997 RSA Laboratories Seminar Series. Proceedings*, 1999.

[16] Rogaway, P., Matyas, S. M.: *RSAES-OAEP Encryption Scheme Algorithm specification and supporting documentation*, 2000.

[17] Solovay, R., Strassen, V.: A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[18] Švenda, P., Nemec, M., Sekan, P., Kvašňovský, R., Formánek, D., Komárek, D., Matyáš, V.: The Million-Key Question – Investigating the Origins of RSA Public Keys. In *25th USENIX Security Symposium. Proceedings*, 2016.

Replicate your identity management

Jan Pazdziora

E-mail: jpazdziora@redhat.com

1 High availability of infrastructure services

When planning and deploying pieces of infrastructure that affect performance or core functionality of large number of machines and services in computer network, it is essential to aim for high availability, as well as for efficient use of resources. It is often helpful when the software itself provides admins with clear and easy way to view and manage the more complex topologies of redundant services.

FreeIPA project is an integrated umbrella for multiple identity management related solutions like directory server, Kerberos Key Distribution center (KDC), certificate system, DNS server, and others. It can be used to manage identities from users, hosts, or services to DNS records, and access control policies for Pluggable authentication modules (PAM) or sudo. Operation of client machines and services can be tighly dependent on the availability of the IPA server, even if the client-side System Security Services Daemon (SSSD) is built around caching and can alleviate some of the impact of unavailable server. Still, when new user logs in for the first time or system is accessed for the first time, their identities (via Name Services Switch (NSS) or DNS) need to be looked up and resolved, and authenticity and authorization of the user for given service verified. Keeping IPA server available and accessible without network latencies is thus important.

FreeIPA has been built around multi-master replication provided by the 389 Directory Server. The complexity of creating the replicas and managing the topology in alignment with overall network needs used to be a hurdle that might have left some deployments in less than optimal state. With FreeIPA releases 4.3 and 4.4, new features were introduced that make much simpler the setup of replicas, management of the replication topologies, as well as control over which servers the client hosts will prefer. Let's look at those features and use cases.

2 Setting up FreeIPA replica

With older FreeIPA versions, creating new replica server involved producing GPG-encrypted replica information file on the original master server using the

Directory Manager password. This file then had to be copied manually to the replica machine and fed to the `ipa-replica-install` command, which again required the Directory Manager password. The need to run operations on the master together with use of credentials which were not otherwise used in standard day to day operation and management of IPA setup posed issues in automated scenarios and provisionings of replicas.

The current versions introduce the notion of promotion of existing IPA-enrolled client to replica, initiated from the client side, typically with admin's credentials. No operation is needed on the master, and gone is the manual copying of the replica information file.

In fact, even the admin's credentials are not needed on the replica machine. Special host group `ipaservers` is used to control the ability of machines to promote themselves to replicas, and the `ipa-replica-install` command can IPA-enroll machine itself, for example using one-time password for the host. If the admin (or some automated provisioning process) creates host entry for new replica, lets the IPA server generate random one-time password, and adds this new host to the `ipaservers` host group, the one-time password can be used on the replica machine to both IPA-enroll it and make it full replica.

One of the possible new workflows therefore can be:

```
client$ kinit admin
Password for admin@EXAMPLE.COM:

client$ ipa host-add replica.example.com --random
----------------------------
Added host "replica.example.com"
----------------------------
  Host name: replica.example.com
  Random password: ImgXN_VxNC,B
  Password: True
  Keytab: False
  Managed by: replica.example.com
client$ ipa hostgroup-add-member ipaservers --hosts=replica.example.com
  Host-group: ipaservers
  Description: IPA server hosts
  Member hosts: master.example.com, replica.example.com
-------------------------
Number of members added 1
-------------------------

replica# ipa-replica-install --password 'ImgXN_VxNC,B'
Configuring client side components
Client hostname: replica.example.com
Realm: EXAMPLE.COM
```

```
DNS Domain: example.com
IPA Server: master.example.com
BaseDN: dc=example,dc=com
...
Enrolled in IPA realm EXAMPLE.COM
Created /etc/ipa/default.conf
...
  Configuring directory server (dirsrv). Estimated time: 1 minute
  [1/43]: creating directory server user
  [2/43]: creating directory server instance
...
  [28/43]: setting up initial replication
Starting replication, please wait until this has completed.
Update in progress, 6 seconds elapsed
Update succeeded
  [29/43]: adding sasl mappings to the directory
...
  [2/2]: configuring ipa-otpd to start on boot
Done configuring ipa-otpd.
```

We see that the host entry and its host group membership was created from separate IPA-enrolled machine and no commands had to be invoked on the IPA master itself. On the replica machine, single command achieved both the IPA-enrollment, setup and configuration of the IPA server, as well as the replication agreement and configuration.

If the replica promotion starts with already IPA-enrolled machine, we can check that the original configuration which pointed to the master in `/etc/ipa/defaults.conf`

```
[global]
server = master.example.com
xmlrpc_uri = https://master.example.com/ipa/xml
```

gets updated to point to itself since the machine is now proper IPA server, after the promotion has finished:

```
xmlrpc_uri = https://replica.example.com/ipa/xml
```

3 Replication topology management

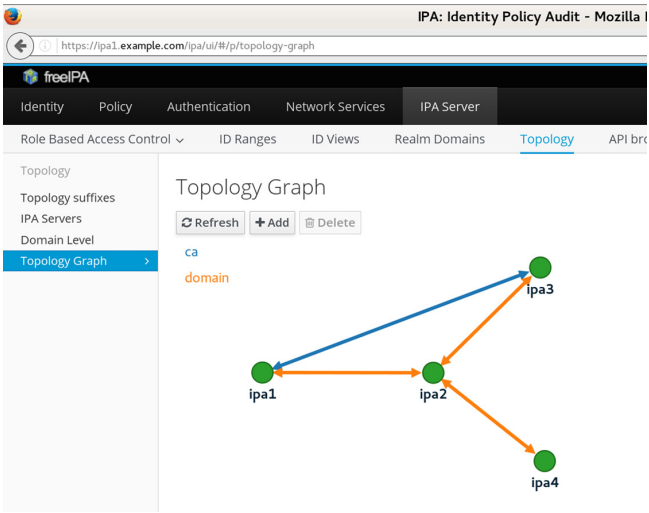
When the first replica is created, it is obvious that there is only one replication agreement between the original master and the new replica. But in larger deployments when there are dozens of IPA server in multiple datacenters on different continents, the way how replication of changes is propagated and how

replication agreements are structured becomes important. The more is not always the better here – as general rule, the number of replication agreements on any given IPA server should not exceed four.

In older versions, it was of course possible to manage the replication agreements but the command line tool `ipa-replica-manage` (and `ipa-csreplica-manage`, for the certificate system replication) had to be run on the IPA server, and the configuration of replication agreement was local to the pair of IPA servers in question.

With latest versions, all information about the replication topology, all the agreements (called segments), is stored in the directory server schema, and thus replicated to all servers in the domain. It is therefore possible to inspect the information with command line tools from IPA-enrolled clients or in WebUI. And the information is not just read-only view of the situation – when segment is created in the directory server, that information is replicated to all servers and when it reaches the target nodes of the new segment, establishment of new replication agreement is triggered. This is especially important when large domain of IPA servers needs to be managed centrally, without the ability to directly interact (with SSH) with the target IPA machines that might be in completely different geographical location. Having a way for replication to reach the remote IPA servers (potentially via chain of other IPA replicas) is enough to setup the replication agreement.

Let us assume situation of four IPA servers. The existing topology can be displayed graphically in Topology Graph in WebUI



or listed with `topologysegment-find` command:

```
ipa1$ ipa topologysuffix-find
---------------------------
2 topology suffixes matched
---------------------------
  Suffix name: ca
  Managed LDAP suffix DN: o=ipaca

  Suffix name: domain
  Managed LDAP suffix DN: dc=example,dc=test
----------------------------
Number of entries returned 2
----------------------------
ipa1$ ipa topologysegment-find domain
------------------
3 segments matched
------------------
  Segment name: ipa1.example.com-to-ipa2.example.com
  Left node: ipa1.example.com
  Right node: ipa2.example.com
  Connectivity: both

  Segment name: ipa2.example.com-to-ipa3.example.com
  Left node: ipa2.example.com
  Right node: ipa3.example.com
  Connectivity: both

  Segment name: ipa2.example.com-to-ipa4.example.com
  Left node: ipa2.example.com
  Right node: ipa4.example.com
  Connectivity: both
----------------------------
Number of entries returned 3
----------------------------
ipa1$ ipa topologysegment-find ca
-----------------
1 segment matched
-----------------
  Segment name: ipa1.example.com-to-ipa3.example.com
  Left node: ipa1.example.com
  Right node: ipa3.example.com
  Connectivity: both
----------------------------
Number of entries returned 1
----------------------------
```

Note that there are actually two replication topologies here – the *domain* for the IPA domain information and *ca* for the certificate system data. We see

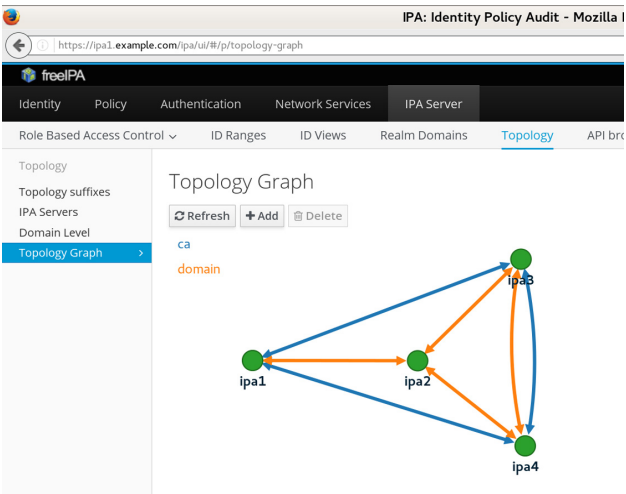
that only **ipa1** and **ipa3** run the certificate servers and have direct replication segment for that purpose, while the domain data is replicated via a "central" node **ipa2**.

If we'd like to add new segment (replication agreement) between servers **ipa3** and **ipa4**, we can do so from any server, via WebUI or via command line tool. On the command line, we could use

```
ipa1$ ipa topologysegment-add domain
      ipa3.example.com-to-ipa4.example.com \
    --leftnode=ipa3.example.com --rightnode=ipa4.example.com
----------------------------------------------------------
Added segment "ipa3.example.com-to-ipa4.example.com"
----------------------------------------------------------
  Segment name: ipa3.example.com-to-ipa4.example.com
  Left node: ipa3.example.com
  Right node: ipa4.example.com
  Connectivity: both
```

If we refresh the Topology Graph on the WebUI, the new line between **ipa3** and **ipa4** will be shown there as well.

It is worth noting that segment can be added only between nodes that already act in given role. For instance, if we want to add replication agreement between **ipa3** and **ipa4** for the *ca* suffix for the certificate system, we first have to enable and configure the certificate system server on **ipa4** with `ipa-ca-install`. That will establish replication agreement with the currently configured CA host (**ipa1**) and only then we can add the next segment. Running `ipa topologysegment-find` or refreshing the page with the Topology Graph in WebUI would confirm our new replication topology:



4 DNS-based locations

So far we have focused on replica setup and management on the server side. We can easily create replicas and tune replication topology. But how are the client machines able to use the replicated setup?

In typical scenarios, IPA-enrolled Linux machines will use `ipa-client-install` to configure various components of the operating system. That configuration can explicitly name a particular IPA server, or it can rely on DNS SRV records to be used to discover the correct service endpoint to use. In case of SSSD, the `ipa_server` can force the SRV lookup with `_srv_` token, and it can be mixed with hostname as well:

```
[domain/example.com]
ipa_server = ipa1.example.com, _srv_
```

will prioritize `ipa1.example.com` and fallback to any other server is can find via DNS. In older versions of FreeIPA, this was the primary mechanism available for "pinning" IPA clients to a particular close (in terms of smallest latency) server.

The problem with this solution is its client-side nature. If additional IPA server is installed in local datacenter, all clients might suddenly need the new server listed in addition to the existing `ipa1.example.com`:

```
[domain/example.com]
ipa_server = ipa1.example.com, ipa2.example.com, _srv_
```

The order of those hostnames would likely mean that the first server will be used by vast majority of clients while the second one will only be utilized when clients cannot reach the first one – hardly a fair load balancing. Other services and libraries might not even support mixing SRV DNS mechanism with explicit hostnames, making localized behaviour setup even harder.

To solve this problem, FreeIPA 4.4 introduces DNS-based locations. They allow grouping of IPA servers and assigning priorities to them, per individual locations.

The basic expectation is that in every network subset where clients are supposed to primarily use certain set of IPA servers, there is at least one IPA server (replica) running embedded DNS server, and that clients in that network subset are configured to use that DNS server. This part of the configuration is outside of IPA's handling – typically DHCP in individual network segments will be configured to provide local DNS server IP address first, and it will either be directly the IPA DNS server, or local DNS server which queries the IPA DNS server during recursive resolution.

IPA DNS server which is assigned to certain location (for example `emea`) will autogenerate location-based CNAME records for certain names: `_kerberos.`

_udp.example.com, _kerberos._tcp.example.com, _ldap._tcp.example.com, ... they all will return respective name under emea._locations.example.com. Those SRV records will in turn contain priorities for individual IPA servers based on their membership in given location, and weights enterred by admin.

Let us see a simple emea location configuration:

```
$ ipa location-show emea
  Location name: emea
  Servers: ipa1.uk.example.com, ipa2.uk.example.com
  Advertised by servers: ipa1.uk.example.com, ipa2.uk.example.com
  Servers details:
    Server name: ipa1.uk.example.com
    Service weight: 10
    Service relative weight: 25.0%
    Enabled server roles: CA server, DNS server, NTP server

    Server name: ipa2.uk.example.com
    Service weight: 30
    Service relative weight: 75.0%
    Enabled server roles: DNS server, NTP server
```

When client machine resolves SRV record to autodiscover a service, it will be "redirected" via CNAME anwer to the location-specific SRV record with location-specific priorities.

```
$ dig +short @ipa1.uk.example.com. _kerberos._tcp.example.com SRV
_kerberos._tcp.emea._locations.example.com.
0 10 88 ipa1.uk.example.com.
0 30 88 ipa2.uk.example.com.
50 10 88 ipa1.houston.example.com.
$ dig +short @ipa1.houston.example.com. _kerberos._tcp.example.com SRV
_kerberos._tcp.us._locations.example.com.
50 10 88 ipa1.uk.example.com.
0 10 88 ipa1.houston.example.com.
50 30 88 ipa2.uk.example.com.
```

We see that depending on which IPA DNS server we query, we get priorities 0 for servers in the same location and 50 for servers outside of it.

This way, SSSD only needs to use ipa_server = _srv_ and other components on the IPA client machines do not need to configure explicit hostnames of servers and services of IPA domain. Any modification to the priorities takes place on the server side, and is of course replicated to all IPA servers. This approach also caters nicely to the needs of roaming users within the organization. Depending on where they connect their laptop to the network, they will obtain local DNS server IP address, and will thus resolve the SRV records via the closest location.

5 Conclusion

With features focused on replicated setups and their management, latest FreeIPA releases make large-scale identity management deployments easier and more efficient. There are no longer any excuses for not having well set up FreeIPA redundancy.

References

[1] *FreeIPA project.* https://www.freeipa.org/

[2] *Replica promotion.* https://www.freeipa.org/page/V4/Replica_Promotion

[3] *Managing replication topology.* https://www.freeipa.org/page/V4/Manage_replication_topology

[4] *DNS locations.* https://www.freeipa.org/page/V4/DNS_Location_Mechanism