Antud kirjatüki teemaks olen valinud *stack overflow* tüüpi turvavigade ning nende ärakasutamiseks kasutatavate meetodite kirjelduse, sest kuigi antud veatüüp on eksisteerinud juba aastakümneid, on eestikeelses kirjanduses selle kohta üldiselt vähe infot.

Stack overflow on *buffer overflow* turvavea alamtüüp, mille edasiseks kirjeldamiseks alustan kõigepealt *buffer overflow* veatüübi üldisema kirjeldusega. *Buffer overflow* viga tekib kui programmi mingi osa kirjutab andmeid mällu üle selleks ettenähtud piiride. Üks lihtsamaid näiteid on selles osas nime küsimise näide, kus programm on kasutajalt nime küsimiseks broneerinud näiteks 100 baidi pikkuse mälupiirkonna (arendaja on teinud eelduse, et kellegi nimi ei ole üle 100 tähemärgi pikk), kuid ei kontrolli kasutajalt tuleva sisestuse pikkust. Kui nüüd aga kasutaja sisestab 110 tähemärki pika nime, siis viimased 10 tähemärki lähevad üle piiride ning kirjutavad üle selleks mitte ettenähtud mälupiirkonna. Sellise tegevuse tagajärjed sõltuvad aga juba väga olukorrast:

1. Kui mälu broneerimisel on kasutusel teatav lisatsoon, tänu millele lühikesed *overflow*-d ei avalda mingit mõju, ei juhtu ka süsteemis midagi.
2. Kui programm on küsinud ülejoostava mälu dünaamiliselt (C/C++ puhul *malloc*-i või *new* vahendusel), siis tekib *heap overflow*, mille puhul tekib probleem tavaliselt alles hiljem, sest *overflow* kirjutab piisava pikkuse puhul üle mõningased *heap* struktuurid ning ajab segamini allokeeritud mälude vahelised seosed. Tavaliselt lõppeb selline olukord programmi kokkujooksmisega mälu vabastamisel, kuid mõningatel juhtudel on selle vahendusel võimalik saada ka kontroll programmi üle. Siiski on viimane tehniliselt märgatavalt raskem kui *stack overflow* puhul.
3. *Stack overflow* tekib olukorras, kus mälu piirkond, mille piire ületatakse, asub programmi *stack*-is. Sellisel juhul on *overflow* puhul kaks peamist tulemust, mida ründajad kasutavad:
    1. Esimene variant on olukord, kus *overflow* kirjutab lisaks ettenähtud mälupiirkonnale (tavaliselt massiiv) üle ka mõne teise muutuja/massiivi mälupiirkonna, mille tulemusena muudetakse nende muutujate väärtust, mille muutmise võimalust kasutajal ette nähtud pole.

```
void kysi_andmed()
{
 char nimi[32];
 char program_name[32] = "ProofOfConcept Program";

 printf("Sinu nimi: ");
 gets(nimi);

 printf("%s tervitab sind %s", program_name, nimi);
}
```

Näites toodud programm töötab normaalselt seni, kuni kasutaja sisestatud nimi on väiksem kui 32 baiti (sõltuvalt kompilaatorist võib see number natuke erineda ning kuna muutujate järjekord *stack*-is ei ole alati sama, mis muutujate järjekord algkoodis, siis ei pruugi see kõigil juhtudel üldse nii toimida). Mis aga juhtub kui kasutaja sisestab 40 tähemärki:

```
Sinu nimi: 1234567890123456789012345678901234567890
34567890 tervitab sind
1234567890123456789012345678901234567890
```

Kuna kasutaja sisestas oma nime, mille pikkus ületas selleks broneeritud mälupiirkonna, siis kirjutas see üle ka muutuja *program_name* mälupiirkonna, mis asub mälus muutujast *nimi*

kõrgemal. Kui nüüd aga mõnes muus programmis saab samal viisil ülekirjutada näiteks muutuja *is_admin, root_passwor*d vms, siis on selge, et selline turvaviga võib anda ründajale ligipääsu süsteemile või osale selle funktsionaalsusest.

2. Teine variant on meetod, mida kasutatakse suuremal osal juhtudest *stack overflow* puhul, et võtta kontroll programmi üle. Selle puhul kirjutatakse üle mitte mõni teine muutuja vaid mälu piirkond, kuhu on kirjutatud funktsioonist tagasimineku aadress. See tuleneb viisist kuidas *stack* töötab. Kui mõni funktsioon kutsutakse välja, siis kirjutatakse *stack*-i ka andmed selle kohta, kust peaks kood peale funktsiooni lõppemist töötamist jätkama. Lisaks laieneb *stack* negatiivse mälupiirkonna suunas. See loob olukorra, kus *stack overflow* puhul on võimalik ülekirjutada funktsiooni tagastumise (*return*) aadress.

```
void kysi_andmed()
{
      char pin[5];

      printf("INSERT PIN( MAX 4 characters ):");
      gets(pin);
}
```

Antud näite puhul võib *stack*-i ehitus märgistatud reale jõudmise hetkeks olla umbes selline (mälu aadressid suurenevad vasakule ning stack laieneb paremale), nagu demonstreerib järgnev tabel.

...	**RETURN ADDRESS**				**EBP**				**pin**					...
...														...
...	.44	.43	.42	.41	.40	.39	.38	.37	.36	.35	.34	.33	.32	...

Esimesel real on info, mis antud mälupiirkonnas asub:
- ***RETURN ADDRES***S – aadress, kust programm peale funktsiooni tegevuse lõppemist töötamist jätkab.
- ***EBP*** – üks vaheväärtus, mille funktsioon käivitudes alati *stack*-i kirjutab. Antud kontekstis pole see oluline.
- ***pin*** – on funktsioonis asuv pin massiiv, millesse kirjutatakse kasutaja sisestatud andmed.

Kui nüüd kasutaja sisestab 4 kohalise pini asemel 13 kohalise, näiteks *abcdefghijklm*, siis on tulemuseks:

...	**RETURN ADDRESS**				**EBP**				**pin**					...
...	m	l	k	j	i	h	g	f	e	d	c	b	a	...
...	.44	.43	.42	.41	.40	.39	.38	.37	.36	.35	.34	.33	.32	...

See aga on kirjutanud üle juba ka ***RETURN ADDRESS*** osa, mis tähendab, et funktsiooni lõppedes ei jätkata tööd mitte kohast, kust funktsioon välja kutsuti vaid hoopis aadressil mlkj ehk 0x6d6c6b6a (vastupidine järjekord sest mälu adresseerimisel on kasutusel *little-endian* järjestus). Olgu ka siinkohal mainitud, et antud näide ei pruugi koodi kopeerimisel üks ühele töötada, sest sõltub tugevalt kompilaatorist ning sellele kaasa antud parameetritest.

Kui ründaja on saanud kontrolli *return addressi* üle, siis on edasine üldpildis üpris sirgjooneline. Ründaja üritab kasutada enda sisestatud andmeid masinkoodina ning paneb *return address*-i abil funktsiooni lõppedes käima just selle. Selle jaoks peab *return address* viitama kasutaja kontrolli all olevale aadressile. Antud näites oleks selleks *pin* massiivi algus (loomulikult on see liiga lühike realses elus toimiva *shellcode* jaoks). Kui nüüd kasutaja sisestaks oma nimeks toimiva masinkoodi ja selle lõppu (see osa mis kirjutab üle *return addressi*) pin massiivi aadressi, siis käivitab programm peale funktsiooni lõppemist just selle masinkoodi, mille ründaja talle ette andis.

Kirjeldatu oli reaalses elus toimuva *exploiti* arenduse tugev lihtsustus, sest tavaliselt ei ole kasutaja kontrolli all oleva muutuja aadress ette teada. Samuti on tänapäeval kasutusel *non-executable stack* ja *address space layout randomization* kaitsed, mis takistavad selliseid ründeid veelgi. Siiski jääb üldine mõte *stack overflow* tüüpi rünnakute puhul selliseks ning erinevates kaitsetest möödumised ning rünnete edasiarendused rajanevad siiski kirjeldatud põhimõtetel.